ChatGPT

GENESIS Algorithmic Trading System – Developer Prompt Book

System Architecture Overview (Event-Driven, Modular Design)

The GENESIS trading system is designed with an **event-driven, modular architecture** to achieve institutional-grade performance, scalability, and reliability. All core functions are separated into distinct modules – such as data feed, strategy logic, execution, risk management, and monitoring – each with a single responsibility and clear interfaces. Modules communicate asynchronously via an internal **message bus** or event stream, rather than direct function calls [1]. This decoupling (using a pub/sub broker like ZeroMQ or RabbitMQ) ensures loose coupling: for example, the data feed publishes market tick events that strategy modules consume, strategy modules publish trade signals that the risk engine intercepts and filters, and only then do signals pass to the execution module [2]. An event-driven loop processes incoming data in real time, triggering module actions without blocking other tasks. This design aligns with best practices used by banks and hedge funds, where separate services for pricing, risk, and orders coordinate via an event bus for scalability and fault tolerance [3]. In GENESIS, if one component fails (e.g. the pattern recognition service), the others continue operating, since they're connected by events rather than hard-coded calls [4].

High-Level Architecture: The system can be visualized in layered form. At the bottom, a **Market Data Ingestion** service interfaces with MetaTrader 5 (MT5) to gather live price quotes and historical data. Above that, the **Signal Intelligence** layer processes raw data into enriched trading signals. A **Strategy Engine** uses those signals (and any strategy-specific logic) to decide on trade actions, which are passed to the **Execution Engine** to place orders on MT5. Throughout, a **Risk Management Engine** oversees all proposed trades and positions, enforcing drawdown limits and other constraints before orders execute. A **Macro Synchronization** module informs strategies of scheduled economic events or sudden market regime changes, triggering pauses or adjustments as needed. Cross-cutting these layers are the **Pattern Detection & ML Feedback** module, which analyzes performance and feeds improvements back into signals and strategies, and the **Backtesting** module, which can simulate the entire pipeline on historical data for research and testing. A **GUI/Dashboard** sits on top for real-time monitoring and control, and a **Compliance/Audit Logging** module underpins everything, recording all events and decisions for transparency. All modules are orchestrated by the message/event bus – e.g. the Market Data module broadcasts ticks, strategies publish signals events, risk publishes "approved order" or "blocked" events, etc., forming a reactive event chain from market input to trade execution [5] [6].

Message Bus & Event Flow: The internal event bus not only decouples modules for resiliency, but also naturally creates a timeline of events that serves as an audit trail (market event → signal → risk check → order → fill) [7]. The system uses standardized event messages for interactions; this means new modules or handlers can be added easily by subscribing to relevant events (for example, adding a new strategy module that also listens to market data events). The asynchronous design also allows modules to scale independently – e.g. multiple signal generator processes can consume the same feed events in parallel, and the execution service can run on a separate machine near the broker server for low latency. By

containerizing modules (Docker or similar), GENESIS can deploy each component as a microservice, enabling horizontal scaling of bottlenecks like data ingestion or execution if needed. The open-source MBATS framework is one example of a Docker-based trading infrastructure that inspired this approach. In practice, GENESIS can run the MT5-connected data gateway in one process, strategy and risk logic in another, and the GUI in a separate web app process, all communicating via the event bus – so a crash in the GUI, for instance, won't halt the trading logic.

FTMO Risk Overlay: A critical aspect of the architecture is the **risk/compliance overlay** applied system-wide to meet **FTMO proprietary trading rules** – specifically the 5% max daily loss and 10% max total drawdown limits. Risk management is "baked into" the event flow: every trade signal event passes through the Risk Engine before any order event is sent to execution. The Risk Engine continuously tracks account equity and drawdown so it can inject **kill-switch events** or block orders that would violate FTMO limits in real time. For example, if an incoming trade signal would cause the day's loss to exceed 5%, the risk module will intercept and cancel it (or downsize it) instead of letting it proceed. Similarly, if total drawdown nears 10%, the risk module can trigger a global **halt event** to flatten all positions and freeze trading. These protective measures are handled at the architecture level (not reliant on trader discretion) – a hallmark of institutional systems. The architecture thus ensures that no single module can violate risk rules: even if a strategy tries to over-leverage, the centralized risk layer will catch it before execution.

Telemetry & Performance Monitoring: In addition to functional modules, GENESIS includes a **telemetry framework** that monitors the system's own performance and health in real time (latency, throughput, resource usage, error rates) [8]. Telemetry events are published on the bus just like market events. For example, the system measures the time from receiving a tick to sending an order; if this latency exceeds a threshold (say 100ms), a **TelemetryAlert event** can notify modules to adjust (perhaps pause lower-priority tasks) [9] [10]. This self-monitoring is crucial for high-frequency operation: if the system gets overloaded (e.g. an explosion of ticks during a news spike), it might automatically throttle the least important strategies or stop non-critical logging to maintain real-time speed [9] [11]. The architecture supports stress-testing by allowing simulation of high message loads – e.g. pumping 200% of normal tick rate through the event bus – to verify that modules still meet performance targets without deadlocks [12] [13]. These telemetry checks and stress tests ensure GENESIS can handle extreme scenarios (regulators like MiFID II expect systems to endure 2x peak load safely) [12] [14]. In summary, by combining a modular event-driven design with robust risk controls and telemetry, the GENESIS architecture is **resilient, scalable, and compliant** – an engineering foundation on par with professional trading desks.

Actionable Architecture Guidelines: To implement this design, developers should (1) clearly define module boundaries and data contracts (events schemas) for **Data Ingestion, Strategy Engine, Execution Engine, Risk Manager, GUI/Monitoring**; (2) set up an internal event bus or message queue (e.g. a Python asyncio event loop or external broker) for all inter-module communication; (3) ensure the system can easily switch between **live trading and backtesting** by abstracting data sources and execution endpoints (feed modules can publish historical data events in backtest mode); and (4) enforce separation of concerns so that each module can be developed and tested in isolation. In the following sections, we detail each major module – its role, design, data flows, best practices, MT5 integration points, and how to leverage AI assistants (Claude 4 and GitHub Copilot) to develop and refine the module efficiently.

Module: Data Ingestion (Market Data Feed)

Purpose & Role: The Data Ingestion module is the **gateway for all market data** into GENESIS. It connects to MetaTrader 5 via the official Python API (MetaTrader5 package) to fetch live price quotes, tick data, bar data, and any other market information needed (like spreads or volumes). This module ensures that strategies and other components always have up-to-date market data. In an institutional-style setup, Data Ingestion provides a unified feed that can be consumed by multiple strategies concurrently. It abstracts the details of MT5's data retrieval, so downstream logic can subscribe to a clean stream of market events (ticks or time-bar events) without worrying about API calls or connection issues. It also handles **historical data loading** for backfills – for example, on startup, it might fetch the last N days of price history for indicators or for a backtest run.

Internal Structure & Responsibilities: Internally, Data Ingestion typically consists of a **connection manager** and one or more **data polling loops** or listeners. Upon startup, it **initializes the MT5 terminal connection** with the appropriate credentials and settings (terminal path, etc.). A robust design checks the return value of `mt5.initialize()` and fails fast if connection cannot be established (to avoid running in a half-connected state). Once connected, the module spawns processes or threads to continuously poll data: for example, one loop might use `mt5.copy_ticks_range()` or `mt5.copy_rates_from_pos()` in a short interval (every second or few seconds) to retrieve the latest ticks or bar updates. Each batch of new data is then published as an event (e.g. a `MarketTickEvent` containing symbol, bid, ask, timestamp) on the internal bus. The module may maintain a small buffer or cache of recent data and can perform some light pre-processing – e.g. converting raw MT5 data into Pandas DataFrames or standard Python objects for convenience. If needed, a **symbol subscription** mechanism is used: the module will ensure the desired symbols are selected/visible in MT5 via `mt5.symbol_select(symbol, True)` before requesting data. Data Ingestion might also include an **historical data fetcher** for on-demand queries (e.g. if a strategy requests a specific range of past data or for backtester use).

The module should be **thread-safe and efficient**: if using multi-threading or async, be mindful that the MT5 API is generally not thread-safe – one approach is to funnel all MT5 calls through a single thread (the data thread) and then broadcast results to others, to avoid "trade context is busy" errors. Also, include error handling: if a data call returns `None` or empty (which can indicate no data or a transient API error), the module should log a warning and retry or handle gracefully.

Input & Output Data: Inputs to this module are minimal – mainly configuration (which symbols to track, polling frequency, etc.) and the MT5 connection. The outputs are continuous **streams of market data events**. Depending on design, it could output different event types: tick events for each tick, bar events at the close of each timeframe candle, order book events if depth is needed (though MT5 Python API doesn't provide full depth, mostly just last tick and quotes). The data format should be consistent and lightweight, e.g. a Python dict or custom Event object with fields like symbol, time, bid, ask, etc. For historical data requests, the output might be a block of historical bars (e.g. a DataFrame) delivered to the requester (like the backtester or indicator calculators).

Interdependencies & Synergies: The Data Ingestion module feeds **every other module** that requires market info. The **Strategy Engine** subscribes to these market events to generate signals. The **Signal Intelligence** module may use incoming ticks to update technical indicators or model features. The **Risk Engine** might also listen to price events to mark-to-market open positions (for real-time P/L and drawdown calculations). The **GUI/Dashboard** uses data events to update price charts in real time. Additionally, the

Macro Sync module might use time stamps from data or detect volatility changes from data streams as unscheduled event triggers. Because of these broad dependencies, the data feed must be reliable and as real-time as possible. To avoid slowing down the rest, heavy processing (like indicator calculations) can be offloaded to the Signal module rather than done in the feed. If the data flow is high-frequency (for example, ticks on multiple symbols arriving hundreds per second), consider using a high-performance queue and possibly filtering or downsampling for modules that don't need every tick (for instance, the GUI might only need updates a few times per second, not every tick).

Best Practices: To implement Data Ingestion robustly:

- **Stable MT5 Connection:** Always initialize with explicit parameters (terminal path, login, etc.) and check for success. If `mt5.initialize()` fails, handle it as a critical error (log and exit or retry after a short delay) rather than continuing in a faulty state. Once running, periodically verify the connection with `mt5.terminal_info()` or `mt5.connected()`; if disconnected (e.g. network down or MT5 closed), attempt a clean `mt5.shutdown()` and then reinitialize after a brief backoff. Integration with the risk module is wise here: if the feed is down, perhaps trigger a trading halt until data is back, because trading blind is dangerous.

- **Efficient Polling:** Use the most efficient API calls for your needs. `copy_ticks_range` can fetch tick-by-tick if fine granularity is needed, but it might be heavy if called too frequently. Alternatively, for many strategies, polling `copy_rates_from_pos` for new M1 or M5 bars is enough, which is less data-intensive. Aim for near real-time updates – e.g. in one implementation, calling `copy_rates_from_pos` in an infinite loop with a 1-second sleep provided up-to-the-second chart updates. Ensure you handle the case where no new data is available (the API may just return the last bar again); you can track the last timestamp seen and only publish events when new time > last time.

- **Threading and Rate Limits:** The MT5 API can only handle one call at a time per Python process (calls are queued internally). Avoid launching multiple concurrent data requests; instead, handle sequentially in one loop to prevent `mt5.last_error()` showing "Trade context is busy" or similar. If multiple symbols are needed, you can batch requests (e.g. use `copy_rates_range` for each symbol sequentially within one loop iteration). If performance becomes an issue, consider running multiple MT5 terminal instances (each on different account or server) in parallel – but that's advanced.

- **Adaptive Frequency:** If needed, adjust polling frequency based on market activity. During very quiet periods (e.g. overnight sessions), you might poll less frequently to save resources, whereas during volatile periods poll faster. However, note that for FTMO and similar, you want to be highly responsive at all times, so generally keeping a steady frequent polling is safest.

- **Noise filtering:** The data feed typically passes raw data, but if there's extremely high-frequency noise that overwhelms the system, a simple throttle could be applied – e.g. only emit tick events if there's a significant price change or a few milliseconds gap. Often, though, such filtering is part of Signal processing rather than the feed.

- **Resilience:** Run the data ingestion on a stable server (for live trading, usually a VPS close to broker). The module should be able to recover from minor glitches: e.g., if a temporary network issue causes

data lag, catch exceptions and retry. Also, on graceful shutdown, always call `mt5.shutdown()` to close the connection.

MT5 Integration Points: This module is the primary user of the `MetaTrader5` Python API. Key functions include: `mt5.initialize()` to connect; `mt5.symbol_select()` to enable symbols; `mt5.copy_rates_from_pos()` and `mt5.copy_rates_range()` for historical and streaming bars; `mt5.copy_ticks_from()` or `mt5.copy_ticks_range()` for tick data; `mt5.symbol_info_tick()` for quick snapshot of last tick; and possibly `mt5.account_info()` if data module also monitors account changes (though account info might be handled by risk). The Data Ingestion must also handle the **time synchronization** between MT5 and the local system – MT5 uses server time; ensure all events carry a consistent timestamp (e.g. convert MT5 time to UTC or local). If needed, incorporate an NTP time check or use MT5's time for all event stamps to avoid confusion. Another integration detail: MT5's Python API requires a running MT5 terminal. So this module might actually **launch the MT5 terminal** if it's not already open (by providing the path in `initialize` or manually starting it in portable mode). Always verify that the terminal is connected to the correct broker and account (you can specify login details in `initialize` or ensure the terminal's config is set). Once connected, maintain that session; if connection is lost (e.g. broker server down), handle it as described (attempt reconnection, and signal the rest of system to pause trading via risk engine).

Recommended Prompt Templates (Claude 4 & Copilot) for Data Ingestion: When developing or troubleshooting the Data Ingestion module, AI assistants can be valuable:

- *Code Generation & Refinement:* **"Draft a Python class for a `MarketDataFeed` that connects to MT5 and publishes live tick events. It should initialize the MT5 terminal (with path and login), select a list of symbols, then continuously poll for new tick or bar data. Include robust error handling for connection issues and examples of logging."** – This prompt to Claude or Copilot can generate a scaffold of the data feed module. Ensure to specify details like the data structures for events (maybe ask for a simple event class or use Python `dataclasses`). After getting an initial version, you might refine with: **"Improve the MT5 data polling code to handle empty results and avoid busy-wait loops. Use `time.sleep()` appropriately and check `mt5.last_error()` for errors."**

- *Debugging & Testing:* If you encounter an issue (e.g., no data coming through or an MT5 error code), you can prompt: **"Our DataIngestion thread sometimes stops updating – here's the relevant code and log output… (include code snippet). It shows `initialize failed, error code = ...`. What might cause MT5 initialization to fail and how can we fix or detect it?"** – Claude 4 can analyze logs and suggest likely causes (like incorrect path, already running terminal, missing permissions, etc.). Another prompt: **"I'm getting inconsistent tick timestamps from MT5. Sometimes the events seem out of order. What could be wrong?"** might lead the AI to mention timezones or that `copy_ticks_range` might not return sorted if called too frequently, etc. Use AI to verify your assumptions.

- *Updating & Extending:* If adding a feature, say fetching an economic calendar inside this module (though that might belong in MacroSync module), you could ask: **"How can I extend the DataFeed to also fetch an economic calendar from an API (like ForexFactory) once a day and broadcast event notices? Provide a code sketch."** – The assistant could suggest using Python's requests to an API and then integrating that as a separate thread or periodic task in the module.

- *Performance Optimizations:* **"Profile this DataFeed loop for bottlenecks. It's using `mt5.copy_ticks_range` in a loop. How can I reduce latency? Consider using asynchronous I/O or a producer-consumer model."** – The assistant might suggest using the asynchronous version if any, or switching to receiving ticks via `copy_rates_from_pos` for efficiency. If Python becomes too slow, it may even suggest moving critical parts into a faster language (but likely not needed unless ultra HFT).

When using Copilot in VS Code, you can write a comment like *"# TODO: Reconnect MT5 on failure with exponential backoff"* and let it suggest code – verify with Claude by describing the approach to ensure it meets best practices. Keep prompts focused and iterative, feeding in any error messages or logs to the AI for analysis. By collaborating with AI on this module, you can quickly achieve a reliable data ingestion pipeline that forms the backbone of GENESIS.

Module: Signal Intelligence (Enhanced Signal Generation)

Purpose & Role: The Signal Intelligence module is responsible for **generating high-quality trading signals** and enriching them with context before they reach the strategy logic or execution. In a basic system, a "signal" might just be a raw indicator trigger (e.g. an MA crossover or an oscillator threshold). In an institutional-grade system like GENESIS, **Signal Intelligence adds an extra layer of filtering, confirmation, and annotation** to these raw signals. The goal is to ensure the system only acts on **high-confidence, context-aware signals** rather than every blip in the data. This module can incorporate technical indicators, statistical models, or machine learning models to identify trading opportunities (buy/sell recommendations), but critically it also measures the **quality of each signal**. For example, if a strategy's model issues a "buy" signal, the Signal Intelligence layer might augment it with data like current volatility regime, trend strength, recent performance of similar signals, etc., and might even veto or down-weight the signal if conditions aren't favorable. By doing so, Signal Intelligence serves as a **smart filter** that passes only the most promising trades to the Strategy Engine, thereby preserving the edge and avoiding noise.

Internal Structure & Responsibilities: This module typically has several sub-components:

- **Indicator Calculation & Feature Engineering:** It computes various technical indicators or features from the raw market data (often using the streams from Data Ingestion). This could include trend measures (moving averages, ADX), volatility measures (ATR, std dev), momentum oscillators, order book imbalance (if available), etc. If using machine learning models, it also prepares input features for those models here.

- **Signal Generators:** Based on those indicators/features, one or more models or rules generate raw signals. For example, one generator might be a simple moving average crossover logic that outputs a signal "BUY EURUSD" when fast MA > slow MA. Another might be an ML model (like a classifier or regression) that predicts short-term price movement probability and signals a trade if confidence is high. Each strategy or trading approach can correspond to a different signal generator component within this module.

- **Signal Enrichment & Filtering:** Once a raw signal is generated, the module **enriches it with context tags and runs quality checks**. Enrichment means attaching metadata such as: volatility percentile at the moment, trend strength score, time of day, recent win/loss streak for this strategy,

news event proximity (from Macro module), etc. These tags provide context so that other parts (like execution or strategy engine) can make nuanced decisions (e.g. maybe the execution engine will size the trade smaller if volatility is extremely high). The module can also filter out signals that don't meet certain criteria: for example, if an ML model's predicted edge is below a threshold, or if a momentum signal occurs in a flat market (context mismatch), it may suppress that signal entirely. High-end systems use techniques like **adaptive noise filters** (Kalman filters, moving average filters) on the price data or indicator outputs to smooth out random noise – ensuring that a minor blip doesn't trigger a false signal. By **reducing noise**, only the more substantial movements generate signals, which has been shown to improve strategy Sharpe ratios by avoiding many small false trades.

- **Signal Monitoring & Learning:** This sub-component tracks how signals perform over time. It records whether signals resulted in profitable trades or not (the **Information Coefficient (IC)** of signals – correlation of signal predictions to outcomes). If it notices that a particular signal type or strategy's signals have been underperforming (IC dropping), it can flag them as low-quality. It might then adjust thresholds or advise the Pattern/ML Feedback module to retrain or tweak that signal generator. This creates a feedback loop to avoid persistent use of a signal that has "lost its edge".

Inputs & Outputs: Inputs include live market data (from Data Ingestion) and possibly macro events (from Macro Sync, e.g. knowing that an NFP report is in 2 minutes might cause signals to be temporarily invalidated). It may also ingest **historical data** for indicator warm-up or for ML model feature windows (e.g. need last 100 bars to compute an indicator). Another input is configuration for each strategy's signal logic (like indicator periods, ML model parameters, etc.). The primary output is **enriched trade signals**. A signal can be represented as an event or object containing fields: strategy or signal ID, action (buy/sell), instrument, confidence score, any suggested parameters (entry price, stop, take-profit if known), and context metadata tags (volatility, trend, etc.). These outputs are consumed by the Strategy Engine (or possibly directly by Execution if using a simplified pipeline, but usually Strategy Engine aggregates or selects signals). In some designs, the module could output **multiple candidate signals** which the Strategy or Portfolio module then picks from or combines.

Interdependencies & Synergies: Signal Intelligence sits between raw data and actual strategy decisions, so it closely interacts with both sides. It relies on **Data Ingestion** for accurate and timely inputs. It may also use **Pattern Detection & ML Feedback** results to refine signal generation – for instance, if the Pattern module suggests that a certain pattern yields false positives on Mondays, the Signal module might incorporate that as a rule (filter out Monday signals for that pattern). Conversely, it supplies the Pattern module with rich data about each signal (outcomes, context) so the latter can learn. The **Risk Engine** might also interface here: some risk checks are done pre-trade, so Risk might subscribe to signals too. For example, risk could modify a signal by attaching a max allowed size given current exposure or even cancel a signal if it would violate exposure limits (though typically risk acts at execution time). **Macro Sync** synergy: the signal module can query macro state – e.g., if the macro module says we are in a "do not trade" window, the signal generator can automatically suppress signals during that time (or mark them as dormant). **Execution Engine** synergy: The enriched info from signals (like "confidence level" or "volatility regime") can be used by execution to choose how to place the order (e.g., high confidence -> use limit order patiently, low confidence -> maybe require extra confirmation or smaller size).

Best Practices: To build institutional-grade signal processing:

- **Contextual Enrichment:** Always tag signals with relevant market context features. For example, when a breakout signal triggers, attach the current volatility percentile and perhaps a trend indicator reading. If volatility is extremely low, the system might treat the breakout signal as less credible (maybe it's a false start). If a mean-reversion signal triggers in a strongly trending market, that context suggests higher risk, so perhaps that signal should be ignored unless trend strength falls. These *regime filters* mimic what human traders do – adjust strategy based on whether market is trending, ranging, volatile, etc..

- **Noise Reduction:** Apply filtering to raw inputs. This can be done via technical means (e.g., require that an indicator's move exceeds a threshold to count as a signal, or use a smoothed price series). **Adaptive moving averages or Kalman filters** can effectively extract the underlying price trend by filtering out high-frequency jitter. An example: instead of using raw tick prices for a signal, use a short-term moving average of price – this avoids reacting to every tick and only reacts when price moves consistently. High-frequency trading systems commonly use such digital signal processing to focus on true signals.

- **Adaptive Signal Thresholds:** Make signal generation parameters dynamic when possible. If the market's volatility doubles, a fixed threshold (say RSI > 70) might trigger too often; maybe require RSI > 80 in high vol regimes. Similarly, if a strategy has had 3 losers in a row, you might temporarily raise the bar for the next signal (require extra confirmation) – this concept of *dynamic gating* can prevent over-trading during a drawdown. Some of these ideas overlap with the Pattern/ML module (meta-models), but the Signal module can implement simpler rules for immediate effect.

- **Information Coefficient Monitoring:** The module (or an associated analytics process) should track how predictive each signal type is over time. If a particular entry signal historically had a 60% win rate but lately is down to 40%, that's important. A common metric is the **hit rate** or average profit per trade of signals in recent history. If quality degrades, either adjust the signal logic or flag it to stop trading. Hedge funds often have an automatic mechanism to disable strategies when their performance deviates significantly from expectations – GENESIS can incorporate a simpler version: e.g., "if this signal produced 10 consecutive losing trades or the P/L is down X over Y trades, disable it and alert for review."

- **Smart Order Routing Info:** Although actual order execution is later, some intelligence can reside at signal level, especially in multi-venue scenarios (for GENESIS+MT5, all orders go to the MT5 broker, so less applicable). But if, say, GENESIS was extended to multiple liquidity sources, the signal could come with a suggestion of venue or method (this is what banks' **SOR – Smart Order Routing** does). Even within MT5, one could tag a signal "urgent" or "normal" which the execution engine uses to decide between a market order or a sniper limit order.

- **Separation from Strategy:** Keep the Signal module focused on whether a trade *should* be made (and why), not on how much to trade or actual portfolio considerations – those aspects belong to Strategy or Risk. The output should be something like "Buy EUR/USD now (signal strength 0.8)". The Strategy Engine or Execution will decide position sizing and actual order placement given risk constraints.

MT5 Integration Points: Signal Intelligence is primarily an analytical layer and may not call MT5 API directly, except perhaps to get certain data. For example, if using some MT5-specific indicators not easily replicated in Python, one might use `mt5.copy_rates_range` to get historical bars for an indicator calculation. But more often, you'd use Python libraries (e.g. TA-Lib, pandas TA, etc.) on the data provided by Data Ingestion. The key integration is ensuring that the signals align with tradeable MT5 symbols and that any instrument-specific context (like contract size, pip value) is accounted for if needed in computing signal thresholds. Also, since MT5 environment might have certain quirks (like different spread at different times), the signal logic could integrate such info – e.g., maybe skip signals when spread > X pips (since MT5 brokers can widen spread around news). The actual trade execution via MT5 will happen later, but the signals might include parameters that feed into the MT5 order (like recommended stop-loss distance).

Recommended Prompt Templates for Signal Intelligence:

- *Code Generation & Refinement:* **"Generate a Python function `compute_signals(data_window)` that returns trading signals for a momentum strategy. It should calculate a 50-period and 200-period moving average from `data_window` (OHLCV data) and output a buy signal if 50 > 200 with upward slope and a sell if 50 < 200. Include a volatility filter: only signal if current ATR is below a threshold. Also add context tags (volatility_level, trend_strength)."** – This prompt would get an initial code from Copilot or Claude. You can then refine by asking for additional features: **"Now modify it to not signal at all during a scheduled news event (pass in a parameter `news_upcoming` and skip signal if True)."** This way, the assistant helps integrate macro context easily.

- *Debugging & Testing:* If signals are too frequent or not behaving, you might prompt: **"My signal generator is giving too many false signals in choppy markets. Here is an example series of prices and the signals it produced… (provide snippet). Suggest improvements to reduce false positives."** The AI might suggest adding a trend filter or requiring confirmation over multiple bars. Another debug prompt: **"I enriched signals with a 'confidence score', but I'm not sure how to calculate an Information Coefficient properly. How can I measure if my signals correlate with future returns?"** – The assistant can explain how to compute IC (e.g. pearson correlation between signal predictions and outcome) and maybe provide code to do it.

- *Updating & Extending:* **"We want to incorporate a machine learning model into Signal Intelligence. Suggest how to integrate a trained XGBoost model that predicts 1-hour price move into the signal generation. What features should we provide and how to use the model's output as a trading signal?"** – This prompt gets design suggestions: perhaps to compute features (like recent returns, technical indicators) and get a probability from XGBoost, then signal buy if probability > 0.7, etc. The AI might also mention caution like avoiding overfitting and retraining periodically. You can then have it draft the integration code or pseudo-code, which you refine.

- *Performance Optimization:* **"The signal processing is taking too long when many indicators are used. How can I optimize calculating multiple technical indicators on a sliding window of data? Can I vectorize or reuse computations?"** – The assistant might suggest using libraries like NumPy/pandas to compute many indicators at once rather than Python loops, or using caching for overlapping computations. If using Copilot, writing a comment like *"# Optimize: avoid recalculating full indicator on each tick, only update last value"* can prompt it to generate code using rolling calculations.

By leveraging AI assistance for the Signal module, you ensure your signal logic is both sophisticated and well-tested. Claude with its 100k context can even take in a large chunk of historical data and your signal outputs to analyze patterns or suggest improvements – essentially acting as a quick research collaborator. Always validate AI-suggested signals on historical data (via the Backtester) to confirm that they indeed improve performance before deploying.

Module: Strategy Engine (Decision & Portfolio Logic)

Purpose & Role: The Strategy Engine module is the **brain that decides when and how to trade** based on the inputs from Signal Intelligence. If Signal Intelligence answers "What signals are telling us to trade?", the Strategy Engine answers "Which signals do we act on, in what combination, and with what position sizing?". In a simple single-strategy bot, the Strategy Engine's role might seem minimal (just follow the signal). But in an institutional-grade, multi-strategy system, this layer is crucial for **orchestrating multiple signal sources, managing overall portfolio exposure, and implementing strategy-specific rules** that go beyond individual signal logic. It acts as a coordinator that can merge or prioritize signals, manage **position states** (e.g., ensure we don't open conflicting trades), and decide **position sizing** according to risk parameters. Essentially, the Strategy Engine turns signals into actionable trade **orders** or target positions that it then forwards to the Execution Engine.

Internal Structure & Responsibilities: The Strategy Engine can be structured in various ways depending on system complexity:

- **Strategy Instances or Classes:** Often each trading strategy (e.g., "Mean Reversion EURUSD" or "Momentum XAUUSD") is implemented as a separate class or component within this module. Each strategy instance will subscribe to relevant signals and market data, maintain its own state (like whether it currently has an open position, last entry price, etc.), and generate trade decisions. This separation allows multiple strategies to run in parallel without interfering, and you can easily enable/disable strategies.

- **Signal Routing & Synthesis:** The module receives **enriched signals events** (with strategy IDs or types). It must route each signal to the appropriate strategy logic. For example, if a signal comes labeled "Strategy: Momentum1 – BUY GBP/USD", it triggers the Momentum1 strategy handler. In some cases, multiple strategies might get the same raw signal feed but interpret differently. If two different strategies produce contradictory signals on the same instrument, the Strategy Engine could have logic to reconcile that (or allow both to trade, but risk module would handle net exposure).

- **Trade Decision Logic:** For each strategy, this includes deciding *if* to act on a signal and *how*. The simplest is one-to-one: every signal leads to a trade. But more advanced logic: e.g., confirm that no conflicting higher-priority strategy is active, or use additional checks like a secondary indicator confirmation (some might call this part of signal intelligence, but often implemented in the strategy code if it's very strategy-specific). The strategy also sets **trade parameters**: such as entry type (market or limit – though Execution might override), stop-loss and take-profit levels (based on strategy rules or ATR etc.), and how much to trade (position sizing).

- **Position Sizing & Portfolio Coordination:** Using risk guidelines, the strategy computes the order size. It might use a fixed fraction of equity (e.g. 1% risk per trade), volatility-adjusted size (bigger positions in quieter markets, smaller in volatile), or Kelly criterion for optimal f. The Strategy Engine

either calculates the lot size to order or calculates a desired portfolio target (e.g., "go long 50k EURUSD"). It should consult the **Risk Engine** constraints – some designs call a function from Risk module here to get the max allowed size given current exposure. The strategy also must account for existing positions: e.g., if a new buy signal comes for EURUSD but we're already long from a previous signal, strategy logic may choose to **pyramid** (add to position) or ignore the signal if we're at capacity. Similarly, if a sell signal comes and we're long, strategy might decide to close or reverse the position rather than open a fresh short.

- **State Management:** The Strategy Engine typically keeps track of running P/L for the strategy, the number of trades taken, last trade time, etc. This helps in implementing cool-down periods (e.g., after X losses, stop trading for a while – a capital preservation tactic) or avoiding rapid-fire entries (to prevent entering multiple times in same bar, etc.). It might also incorporate **session management** – e.g., only allow trades during certain hours or limit trades per day (which could be considered risk rules as well).

- **Output:** The final output of Strategy Engine is a **trade order request** or **target position** that is passed to Execution. This typically includes instrument, direction (buy/sell), volume, and possibly desired entry price (if using limit) and stop/take-profit. The strategy engine might set a *soft target* (like "I want to be long 100k units"), and the execution module will figure out how to achieve that (maybe all at once or scaled in). More commonly, it sends discrete orders like "buy 0.5 lots now at market, with SL=..., TP=...".

Interdependencies & Synergies: The Strategy Engine is at the center of a web: it interacts upstream with **Signal Intelligence** (consuming signals) and downstream with **Execution** (sending orders). It also heavily interacts with **Risk Engine** – possibly querying risk for allowed size or receiving risk alerts (e.g., risk might send an event "StopTrading" if daily loss limit hit; the strategy engine then must halt new trades). If the system is multi-strategy, the Strategy Engine also implicitly interacts between strategies: they might share the same capital pool, so this module (or risk) must ensure the total exposure remains in check. For instance, if Strategy A is long 5 lots EURUSD and Strategy B also wants to long EURUSD, together they might exceed risk limits even if individually they're fine – a robust design coordinates this (often risk engine does at order time, but strategy engine could also incorporate logic to avoid overcrowding one trade). Another synergy is with the **Macro Sync**: the Strategy Engine should respect macro events – if Macro module signals "pause trading", the Strategy Engine should temporarily not generate orders (or explicitly cancel signals). The **Pattern/ML Feedback** module can also feed into Strategy Engine by adjusting strategy parameters or enabling/disabling strategies. For example, if Pattern detection says "Strategy A's edge is decaying", the Strategy Engine could automatically reduce the trade size for Strategy A or turn it off until further notice (with appropriate logging). Implementation-wise, that could be done by Pattern module sending an event that the strategy engine listens to and marks that strategy inactive.

Best Practices: Key practices for a professional Strategy Engine:

- **Separation of Strategy Logic:** Keep each strategy's code modular and independent. This is similar to QuantConnect's Algorithm Framework where Alpha Model, Risk Model, etc., are separate. In GENESIS, define an interface for a Strategy (e.g., each strategy class has methods like `on_signal(signal)` and `on_tick(data)` and it can output order requests). This allows plugging in new strategies easily without altering the core engine, and if one strategy has a bug and

crashes, it doesn't take down others ideally (if separated by threads or processes). It also eases testing – you can unit test a strategy's logic with simulated signals.

- **Unified Portfolio View:** If multiple strategies trade the same account, maintain a unified view of current positions across strategies. Whether the Strategy Engine itself tracks this or queries MT5 via Execution, it's important to avoid scenarios like two strategies unknowingly trading against each other. Some implementations have a **Portfolio Manager** sub-module that aggregates all desired positions from strategies and resolves conflicts (like a simple rule: net positions in the same symbol should not exceed X or if two strategies disagree, maybe prefer the higher priority strategy's view). This adds complexity, but clarity here prevents unintended risk.

- **Backtest and Live Consistency:** Design the Strategy Engine such that the **same code can run in backtest and live**. This means avoid direct calls to MT5 in strategy logic; instead, abstract order placement (call a method that in live passes to execution, in backtest simulates it). Also, ensure use of event-driven paradigm so that feeding historical data events triggers the strategies just like live ticks do. This parity was emphasized in the architecture principles.

- **Hard Rule Enforcement:** While Risk Engine is the final guardian, Strategy Engine can implement certain risk rules at strategy level too, as an extra layer. For example, a strategy might have a rule "no more than 3 concurrent trades" or "if two losses in a row, skip the next signal". These protect the strategy's performance and help adhere to overall risk discipline. Always implement these deterministically and document them, as they affect trading behavior significantly.

- **Logging and Transparency:** The Strategy Engine should log its decisions: when it accepts a signal and creates an order, log which signal (ID and context) caused it, what size, etc. If it ignores a signal (due to conflict or cooldown), log that too ("Signal at 10:30 ignored by Strategy B due to existing position"). This is invaluable for debugging and compliance – after the fact, you can explain why a trade was or wasn't taken [15].

- **Parameterization:** All strategy parameters (indicator periods, risk per trade, etc.) should be configurable (perhaps via a JSON or config file), not hardcoded. This makes it easier for prompt engineers or developers to adjust strategies without digging into code, and it also ties into Pattern/ML module for auto-tuning – that module could update a config file or call a method to tweak a parameter, which Strategy Engine then uses (maybe after a safe verification).

MT5 Integration Points: The Strategy Engine typically does not call MT5 directly (that's Execution's job). However, it might need to be aware of account or position info from MT5. For example, to know current open positions and profit, it could either track orders it sent (and confirm fills via Execution) or occasionally query `mt5.positions_get()` or `mt5.account_info()`. If Execution is separate, it might be better that Execution broadcasts any fill events or position updates which Strategy Engine listens to and updates its state. For instance, if an order is filled, strategy sets its internal flag "currently in trade" etc. The Strategy Engine may also interface with MT5 for **account equity** if needed for sizing (though Risk or Execution could provide that). Ideally, rely on the single source of truth: Execution can provide a callback or event when a trade is executed or closed, carrying P/L info and new position sizing, which strategy uses.

One integration nuance: if using MT5 hedging mode (multiple positions per symbol), strategies must be careful. Many retail algos run on netting mode (one position per symbol). If in hedging mode, two

strategies could open opposite positions on the same symbol and MT5 would keep them separate (rather than netting them out). This can complicate risk calculations. Decide early if you use netting (more straightforward, one overall position per symbol) – likely in FTMO accounts, they use **netting mode** on MT5 for FX, which simplifies oversight. The Strategy Engine then should assume any trade on an already-held symbol will modify that position (increase, flip, close). Ensure the strategy logic accounts for that (it might check `if position exists: adjust or exit instead of new entry`).

Recommended Prompt Templates for Strategy Engine:

- *Code Generation & Refinement:* **"Generate a Python class `StrategyEngine` that can manage multiple strategies. It should have a method `register_strategy(strategy)` and handle incoming signal events by dispatching to the appropriate strategy's `on_signal`. Provide a simple Strategy base class with an example implementation that logs signals and returns an order."** – This can scaffold a plugin architecture for strategies. Follow-up: **"Extend the Strategy base class to include a position sizing method that calculates lot size based on a fixed percent risk (e.g., 1% of equity with a given stop-loss distance)."** – The AI can write a formula for that (maybe using account balance passed in).

- *Debugging & Testing:* Suppose a strategy is not taking trades as expected. Prompt: **"Our MomentumStrategy did not send any order even though a valid signal was received. Here is the relevant log snippet and code… (include them). What conditions might be preventing the trade, and how to fix or further log it?"** The assistant might notice from code that maybe `self.in_position` flag wasn't reset, or a time filter prevented trading at that hour. It could suggest adding more verbose logs or adjusting that logic. Another debug: **"Two strategies both went long on the same symbol, causing an oversized position. How can I modify StrategyEngine to prevent this? (for example, only one strategy per symbol active at a time)."** – This prompts suggestions like a global symbol lock or letting risk reject the second trade (though proactive prevention is possible too).

- *Updating & Extending:* **"Add functionality to pause all strategies upon a `TradingHalt` event from the Macro layer. Provide a code modification where StrategyEngine listens for a `TradingHalt` event and sets a flag to ignore new signals, and also a way to resume."** – The AI might show using an event handler or a simple boolean that strategies check. Also, **"Implement a cool-down: after a strategy experiences 3 consecutive losses, it should stop trading for 1 hour. Suggest how to track this and enforce it."** – The assistant can outline maintaining a loss counter and timestamp, and skip signals if cooldown active.

- *Performance Optimization:* **"We have 10 strategies and performance is slowing down. How can we parallelize strategy processing? Should we use threads or async for running strategies concurrently on incoming events? Provide an example of how to implement this safely."** Claude or Copilot might suggest using Python threading or multiprocessing with care to not duplicate data feed. It may highlight Python GIL issues and possibly using asynchronous event loop if strategies are I/O bound. If concurrency is implemented, definitely test thoroughly to avoid race conditions. The AI can help identify potential thread-safety issues, e.g., "if two strategies try to modify the same position data simultaneously."

Using AI tools, one can rapidly iterate on strategy logic – from drafting new strategies to verifying that changes still satisfy constraints. With Claude's large context, you could even feed in a sequence of trades from a backtest and ask, *"What patterns do you observe in strategy performance? Should we adjust stop-loss values?"* – acting as a pseudo quant analyst. Keep prompts clear about the rules you want to enforce, and always double-check AI-proposed logic (especially around financial calculations) for correctness.

Module: Execution Engine (Order Management & Low-Latency Execution)

Purpose & Role: The Execution Engine is tasked with **converting trade decisions into actual market orders** on the MT5 platform, doing so in an efficient, controlled, and intelligent manner. It sits at the end of the decision pipeline, as the last gate before trades hit the market. A well-designed Execution Engine seeks to **minimize slippage, respect all broker constraints, and execute orders in a way that achieves the best possible price given the strategy's intent**. In institutional terms, this is analogous to an **Order Management System (OMS)** combined with **Execution Algorithms**. It manages the mechanics of order placement: choosing order types (market vs limit), handling partial fills, managing stops and take-profits, and ensuring that no execution violates risk or compliance rules (like max daily loss). The Execution Engine must be highly reliable and fast, as any delay or error here can directly cost money or break rules.

Internal Structure & Responsibilities: Key components of the Execution Engine include:

- **Order Receiver/Queue:** It accepts order requests from the Strategy Engine (or directly from strategies). These might come in asynchronously, so the engine typically queues orders and processes them sequentially or based on priority. This ensures that if multiple signals arrive at once, it can handle them one by one (or in parallel if independent and thread-safe).

- **Pre-Trade Risk Check:** Before executing, the engine performs a **final check with the Risk Engine** – e.g., call a `risk.allow_order(order)` function. Even if strategy did sizing, this is a belt-and-suspenders to prevent any rule breach. If Risk says no (because this trade would break the daily limit or other rule), Execution will **abort the order** (and log the rejection). This is crucial for FTMO rules: *the execution engine is effectively the last line of defense* to ensure we never place a trade that could cause >5% daily loss or >10% total drawdown if it hit its stop. Concretely, Execution can compute potential loss = order_volume * stop_loss_distance * pip_value; add that to current day loss and see if > 5% equity – if yes, do not execute.

- **Order Construction:** The engine builds the actual MT5 order request (using the `mt5.order_send()` function). This involves specifying symbol, order type (market, limit, stop, etc.), volume, price (if limit/stop), stop-loss and take-profit levels, and slippage tolerance (deviation). It must decide on appropriate **order type and parameters based on context**. For instance, for a normal entry, it might choose a limit order at a better price instead of a market order, to reduce slippage ("sniper-style" execution). The engine might implement logic like: if we got a buy signal with current price 1.1000, rather than instant market buy, place a buy limit at 1.0998 (slightly below) and wait briefly. Or if urgency is high (like a stop-loss exit or a fast breakout), it may use a market order with a small slippage allowance. These tactics are often configurable. The module might have multiple execution algorithms: e.g., *Passive mode* (use limit orders), *Aggressive mode* (use market or stop orders to enter immediately), *Pegged mode* (like placing limit orders that follow the price).

- **Partial Fill and Retry Logic:** Especially with limit orders or in fast markets, not all orders fill instantly. The engine should handle the case where an order is partially filled or not filled at all within some time. It could decide to **cancel or adjust** unfilled orders. For example, if a sniper limit order doesn't fill within 5 seconds or price moves too far away, the engine might cancel it to avoid chasing a bad fill later. If partial fill happened (common if trading large size and not all liquidity was available), it might leave the remainder for a moment or re-place it, or just treat the partial as the final (depending on strategy instructions). MT5 has some native behaviors: e.g., an order can be sent with `ORDER_FILLING_RETURN` which will fill as much as available and not cancel the rest. The engine should leverage these where possible (for instance, in MT5 for non-Forex, by default partial fills can occur; `ORDER_FILLING_RETURN` explicitly allows partial fills and returns remaining volume unsatisfied, which the engine can then re-attempt or adjust).

- **Slippage and Latency Management:** The engine is responsible for trying to minimize slippage (difference between intended price and execution price). It sets a **deviation (slippage tolerance)** on market orders – e.g., allow at most 1 pip slippage; if more, the order won't execute. This prevents "bad fills" in volatile spikes. It also measures latency: how long the request took, etc. Ideally, it runs on a low-latency environment (e.g., the code near the broker's server). The engine should be optimized to do minimal work at the critical moment of order sending: any heavy computations (like complex logging or indicator calcs) should not block this thread. Pre-calculate as much as possible (e.g., position size was done in strategy; risk check is quick math; everything ready so `order_send` can be called immediately on signal).

- **Post-Trade Handling:** After sending the order, the engine checks the result. If success, it logs the trade and possibly sets up tracking (like monitor when the trade closes, etc.). If it included stop-loss/take-profit, MT5 will handle those once order is placed (stop-loss is server-side). If the result is failure (e.g., `order_send` returns not executed), handle accordingly: log the error code (could be `TRADE_RETCODE_REJECTED`, `MARKET_CLOSED`, etc.), maybe retry if it's a retry-able error (like `TRADE_CONTEXT_BUSY` might require a slight delay and retry). If order was not executed due to price moved (for a limit), we might decide to try a market order or adjust price if strategy still wants in. Those decisions could be strategy-specific or a general rule (maybe configurable: "if limit misses, chase by X pips then give up"). The engine might also handle **moving stop-loss to breakeven or trailing stops**, although these could also be managed by the strategy or risk module. In many systems, Execution monitors price to move a trailing stop server-side by modifying the order via `mt5.order_send()` (modify).

- **Compliance & Broker Constraints:** Execution must ensure each order obeys not only risk but also technical broker constraints: e.g., not exceeding max lot size (if a trade requires 200 lots but max is 100, split into two orders), not sending too many orders too quickly (some brokers have limits – the engine could throttle if needed). Under FTMO rules or broker rules, avoid prohibited behaviors: e.g., some prop firms forbid news trading or high-frequency scalping beyond a certain rate. If applicable, the engine might enforce a minimum time between opening and closing trades (if the firm disallows very short trades), or a ban on trading during certain forbidden times (some firms disallow trading in first minutes of daily session or around certain news). These specifics depend on FTMO or other prop rules beyond just drawdown – execution should be configurable to comply (e.g., have a "news blackout" mode that Macro triggers). We already plan Macro to pause signals, but Execution is the final checkpoint: if a trade request comes during a forbidden window (perhaps Macro failed to pause), Execution should refuse it with a log "Order blocked: within restricted news window".

Input & Output: Input is trade orders or "trade intents" from Strategy Engine. These could be represented as objects containing symbol, volume, side, etc., plus maybe some execution preferences (like desired style or urgency). Output is actual **order execution results**. The Execution Engine will typically produce events like "OrderPlaced" (with details and order ID) and "OrderFilled" or "TradeOpened" when confirmation comes. It also outputs **Position updates** (either it can query new positions or rely on MT5 trading events to update current positions). It should inform other modules of these outcomes. For example, it can send an event to the GUI and Risk Engine that a new position of X lots was opened on EURUSD at price Y, with stop Z. The Risk Engine will then include that in its tracking of exposure. If an order is rejected or cancelled, that output might go back to Strategy to handle (maybe strategy could try something else or at least mark that no position was opened).

Interdependencies: Execution is closely tied to **Risk Engine** (for pre-checks and for implementing certain risk logic like kill-switch) and to **MT5 API**. It also interacts with **GUI/Monitoring** – e.g., a manual override from GUI might instruct Execution to close all trades (a "flatten" command) – Execution module would then execute that by sending market close orders for each open position. Execution interacts with **Macro Sync** indirectly; Macro can impose restrictions that Execution might check as described. It's also the source of truth for the **Compliance Log** about trades – every order and fill should be recorded (with timestamp, price, etc.) by Execution (or by a logging observer attached to it). If the system needs to be audited, these logs explain exactly what was sent to the broker and when.

Best Practices: For an institutional-grade Execution Engine:

- **Use Limit Orders Strategically:** Don't reflexively market order for entries if not necessary. Placing **"sniper" limit orders** at advantageous prices can dramatically reduce slippage costs. For instance, if you're buying, try to buy on a brief dip instead of crossing the spread immediately. However, always consider the trade-off: being too passive might lead to missed trades. A common approach is a **timeout or chase**: place a limit order around current price; if not filled in X seconds or if price starts moving away by more than Y pips, then convert to a market order or adjust the limit up (chasing). This way you attempt to get a good fill but ensure you don't completely miss a big move.

- **Immediate Exit for Critical Stops:** When it comes to stop-loss or risk-off scenarios, do not use passive execution. For example, if a stop-loss price is hit, send a market order to close (with a modest slippage allowance) because we want out ASAP to prevent further loss. Similarly, if the Risk Engine triggers a **kill-switch** (like drawdown limit exceeded), Execution should *immediately* market-close all positions without delay. Speed and certainty trump getting an extra pip. Ensure the kill-switch code is thoroughly tested (e.g., try closing multiple positions in quick succession, handle any rejections, etc.).

- **Batching and Child Orders:** If trading a large volume that could impact the market, break it into smaller **child orders** rather than one big order. For instance, 50 lots might be split into five orders of 10 lots each, separated by a few milliseconds or sent to different liquidity streams if available (MT5 typically has one liquidity feed, but conceptually). Randomize or vary the child orders to not be predictably all at once (reduces market impact). For extremely large or sensitive strategies, one might implement execution algorithms like VWAP (spread execution over time to match average price) or Iceberg orders (show small portions at a time). FTMO account sizes might not require such heavy-duty tactics, but it's good practice if scaling up.

- **Latency Optimization:** Host the execution logic on a server close to the MT5 trade server (for FTMO, choose a VPS in same region as their server). Within code, avoid any blocking calls in the execution thread. For example, do not do extensive logging synchronously when sending an order; better to queue logs or use async file writes. Use timeouts in `mt5.order_send()` wisely – it will wait for broker execution; setting a reasonable timeout prevents hang. You might also measure round-trip time and log it (to track if execution is slowing down). If using Python and needing more speed, consider using the MQL5 side (EA) to execute or a compiled component for the critical path, but typically Python with MT5 API is fine for moderate frequency.

- **Handling Disconnects & Errors:** The execution engine should be prepared for cases like "trade context busy" (if two calls overlap) – possibly by locking around `order_send`. Also if MT5 loses connection mid-trade, or the order returns `RETCODE_TIMEOUT`, meaning it's not sure if it placed or not – the engine should query positions/orders to confirm. If uncertain, it might not retry blindly to avoid duplicate orders (some logic: if no confirmation, check account history or order history after a few seconds to see if it went through). This is complex but necessary to avoid ghost orders.

- **Adhere to Prop Trading Rules:** For FTMO specifically, aside from drawdown limits, ensure no **forbidden strategies** are accidentally triggered. FTMO disallows strategies like latency arbitrage, grid/martingale beyond reason, or copy trading from a live source. Execution can enforce some of these: e.g., don't send a flurry of 100 orders in a second (could be seen as latency arb or system malfunction). If the strategy tries that (bug or otherwise), Execution should throttle or even block (and trigger kill-switch if something goes wild sending too many orders – perhaps define >N orders per minute as an error condition). Logging these incidents helps prove compliance (e.g. "Throttled 20 orders – potential erratic behavior" in logs).

MT5 Integration Points: Execution is the main user of `mt5.order_send()`. It needs to construct `TradeRequest` structures properly. Use `mt5.TRADE_ACTION_DEAL` for market orders, `TRADE_ACTION_PENDING` for placing pending orders (limits/stops). The module should also use `mt5.OrderCheck()` if available to validate an order without sending (this can catch some errors like volume too low or trading halt). Another important integration is retrieving trade results: after sending, `order_send` returns a result object with things like `order` and `deal` IDs. For market orders, usually you get a `deal` (actual trade) immediately if filled, whereas for pending you get an `order` ID. The engine should handle both. For stop-loss and take-profit, you can include them in the initial order request (preferred, so they're set server-side instantly). Alternatively, one can send separate modify orders to set stops after entry, but that's slower. MT5 allows setting SL/TP in the initial deal request for market orders as long as you provide price fields. Use that to your advantage (less round-trip).

Additionally, Execution might use `mt5.positions_get()` to verify current positions and `mt5.orders_get()` to see open pending orders if needed (like canceling an open limit if conditions change). If implementing trailing stops, Execution might periodically call `mt5.order_send()` with action=MODIFY on an open position to adjust its SL price.

Recommended Prompt Templates for Execution Engine:

- *Code Generation & Refinement:* **"Write a function `execute_order(order_request)` that takes an order object (with symbol, volume, side, type, price, sl, tp) and uses `mt5.order_send` to**

execute it. Include logic for retrying on `TRADE_RETCODE_BUSY` and logging success or failure." This prompt yields a baseline execution function. Further, **"Enhance the execution logic to handle a sniper limit entry: if order_request.type is 'LIMIT', place a limit order and implement a timeout thread that will cancel the order if not filled in X seconds."** – The AI might outline using Python threading or a scheduled task to cancel (it might not get all details right, but gives a structure).

- *Debugging & Testing:* If encountering execution issues, ask the AI to interpret MT5 return codes: **"I got TRADE_RETCODE_REJECT and TRADE_RETCODE_INVALID_VOLUME errors from order_send. What are the possible causes and how to fix them?"** – It will explain (e.g., invalid volume might mean volume not a multiple of lot step or below minimum; rejected could mean market closed or not enough margin, etc.). Another: **"We experienced large slippage on an NFP news trade. The log shows the market order filled 10 pips away. How can we adjust the Execution Engine to prevent such slippage in future?"** – It might suggest using pending orders before news, or setting a tighter `deviation` so order won't fill if slippage beyond threshold (though then you risk not executing at all). This is a trade-off discussion AI can help articulate.

- *Updating & Extending:* **"Implement a feature to split large orders. If volume > MAX_LOT (say 50 lots), then break into smaller chunks and execute sequentially. Provide pseudocode or code changes for `execute_order` to achieve this."** The assistant will likely produce a loop that sends multiple orders and possibly waits between them. Also: **"Add kill-switch handling: write a method `flatten_all_positions()` that closes all open trades immediately. It should iterate over `mt5.positions_get()` and send market close orders. Include this in Execution Engine."** – We can get a code snippet to do just that (and double-check that it correctly sets opposite order type to close, etc.).

- *Performance Optimization:* **"Our execution logging to disk is slowing things during high frequency. Suggest how to log asynchronously or batch logs."** – The AI might suggest using a separate logging thread or a message queue for logs. Or using Python's builtin `logging` with an appropriate handler that doesn't flush every time. Also could mention using `mt5.shutdown()` properly on program exit to not hang.

GitHub Copilot in VS Code can autocomplete a lot of boilerplate for sending orders if it's seen similar code. For example, writing `mt5.order_send({` will prompt it to fill in required fields. Use Copilot to speed up writing those structures, but verify each field (symbol, volume, type, etc.) is correct. Claude's advantage is explaining *why* something might fail, which is great when fine-tuning execution logic for edge cases.

Module: Risk Engine (Real-Time Risk Management & Compliance)

Purpose & Role: The Risk Engine is the **guardian of the trading system's capital and rule compliance**. Its core purpose is to **monitor and control risk at multiple levels** – from per-trade risks to aggregate portfolio exposure to hard drawdown limits – ensuring that the system stays within predefined safety bounds at all times. In GENESIS, which must abide by strict FTMO rules, the Risk Engine is particularly crucial: it is responsible for enforcing the **5% max daily loss** and **10% max overall drawdown** limits, as well as any other proprietary or regulatory constraints (max lots, max concurrent trades, etc.). This module operates in both a *preventative* manner (blocking or resizing trades that would violate limits before they happen) and a *reactive* manner (taking action if something unexpected slips through, like closing positions if

a threshold is hit). Essentially, it's the automated risk manager that a human trader or risk officer would be in a hedge fund, making sure the system "does not blow up".

Internal Structure & Responsibilities: The Risk Engine can be thought of as having several layers of checks and controls:

- **Per-Trade Risk Check:** Every potential order goes through here (often by Execution Engine querying it). This check verifies that the trade has a stop-loss and known risk, and that this risk is acceptable. For example, if a trade of volume X with stop-loss Y pips away could lose $Z, ensure $Z is less than the allowed risk-per-trade (maybe 1% of equity or a fixed amount). If the trade lacks a stop-loss, Risk Engine should reject it outright or attach a default hard stop (never allow unbounded risk). It also might enforce that the stop-loss isn't too loose (e.g., if someone tries a 500 pip SL which could break limits). This is also where position sizing can be double-checked – e.g., if the strategy tries to trade 2 lots and the rule says max 1 lot per trade, downsize it or split it.

- **Aggregate Exposure Check:** This looks at current positions plus the new order. Ensure that adding the new position won't exceed limits like max leverage or max exposure per instrument or correlation group. For example, if max allowed exposure is $100k and we already have $80k open, then a new $30k trade would exceed – Risk can either reject it or reduce it to $20k to fit the limit. Exposure limits also include things like "no more than N open trades" or "not more than 2 strategies in the same currency pair" depending on config. This overlaps with strategy engine concerns, but risk is the final enforcer.

- **Drawdown Monitoring (Realtime):** The Risk Engine continuously monitors the account equity relative to starting equity for daily and overall drawdowns. It typically subscribes to account balance updates or calculates unrealized P/L from price feeds. It keeps track of the *peak equity* (highest value) and current equity to compute total drawdown%, and the *equity at start of day* vs current equity for daily drawdown%. If at any point, the drawdown approaches the limit (say we're at 4.5% down on the day approaching 5%), Risk Engine can take preventive actions: it could disable opening new trades (soft halt). If the limit is actually hit or breached, then it triggers the **Kill-Switch**: closing all positions and pausing trading for the remainder of the day (or indefinitely until human intervention). These actions must be *immediate* and logged ("Max Daily Loss hit: closed all positions"). For FTMO, *even open trades count* towards the daily loss if they're in drawdown, so the Risk Engine must include unrealized losses in that calculation. For example, if current open trades P/L = -4% and closed losses = -1%, then we hit 5% even if not all losses are realized; the system should not wait, it must act to cut risk.

- **Kill-Switch Mechanism:** A subcomponent of risk or execution, the kill-switch is a global abort. Risk Engine decides when to invoke it (various triggers: drawdown limits, or detection of some erratic behavior like too many orders, or manual emergency from GUI). When triggered, risk communicates to execution to flatten positions (market close everything) and to strategy engine to halt new signals. It may also set a state "trading halted" until conditions are reset or a human resets it. This mechanism should be very robust – test it thoroughly so that it indeed closes all and doesn't get stuck if one close fails (e.g., if one trade couldn't close, maybe try again or ignore if small).

- **Additional Controls:** These can include:

- **Max Order Size:** e.g., no single order > X lots.
- **Max Slippage Tolerance:** If an order is attempted far off market price (maybe a bug), block it (like if an order price is 5% away from current price – it's likely an error).
- **Cancel-on-Disconnect:** If connection to broker is lost, risk might treat it like a dangerous scenario and flatten positions or at least not allow new ones until reconnect (so we don't blindly hold or trade when we can't see price).
- **Time-based rules:** e.g., no trades after 4:55pm on Friday (to avoid weekend gap risk) – risk can enforce by not allowing new trades near that time and closing positions if required by compliance.
- **Margin Monitoring:** Keep an eye on margin usage; if margin level falls below a threshold, take action (scale down positions) to avoid broker margin call.
- **Logging & Alerts:** Risk Engine logs every significant event: "Order X blocked for exceeding max lots", "Daily loss = 5.1%, all trades closed by system" [16] . It can also send alerts (e-mail or GUI notification) for such events, which is useful in prop trading to notify the trader or risk officer immediately.

Inputs & Outputs: Inputs are: trade proposals (for pre-check), live account data (balance/equity updates, margin), market data (for mark-to-market P/L), and configuration (risk limits, etc.). It might also take input from a **trader UI** (e.g., someone could raise or lower risk limits or manually trigger kill-switch via GUI). Outputs include: **approvals or rejections** back to Execution on orders, **actions** like "close position" commands to Execution, and **status events** like "RiskLimitHit" to notify Strategy or GUI. It also outputs logs for compliance. Sometimes, the risk module might also produce periodic summary metrics (like current drawdown%, current leverage) that GUI can display.

Interdependencies: Tight integration with **Execution Engine** (two-way: Execution queries risk on each order; Risk instructs Execution to kill trades on triggers). Close ties with **Data/Market feed** (for real-time P/L calc – Risk might subscribe to tick events to update unrealized P/L of open positions continuously). Also ties with **GUI**: the GUI might display risk metrics and allow certain risk overrides like an admin turning off trading or changing a limit (if allowed). With **Pattern Detection**: risk data (like volatility of P/L, frequency of near-misses of limits) could be fed into pattern analysis for improving strategies or risk settings. But mainly, Risk Engine stands alongside trading decisions to enforce discipline.

Best Practices:

- **Hard Stops on Every Trade:** Ensure every single trade has a **hard stop-loss** from the moment of execution. If a strategy doesn't provide one, the system should attach a default (like perhaps a catastrophic stop at say 2% of account). Never widen stop-losses beyond initial – Risk Engine could explicitly forbid code from moving a stop further away (only allow closer/tighter adjustments). This prevents the common pitfall of letting losses run.
- **Dynamic Position Sizing:** Encourage consistent risk by adapting position size to market conditions. The risk module or strategy's sizing function should compute volume such that risk % per trade is constant (so larger stop distance = smaller volume). This keeps risk exposures uniform. Risk Engine can double-check such sizing: e.g., recalc expected loss = (entry – stop)/*lots*value per pip, ensure it ~ equals desired risk budget (0.5% equity). If not, adjust volume before allowing order. Also cap size if needed (if account grew but you want absolute cap, or broker max).

- **Continuous Drawdown Tracking:** Implement the drawdown calculation with care. For daily drawdown, track equity at the start of the trading day (which can be a fixed reset time, e.g., FTMO Challenge typically resets at 00:00 CE(S)T). Any drop from that is daily loss. For max overall drawdown, track the highest equity achieved historically (since start or since last reset) and measure drop from there. If possible, include *unrealized* losses in those calculations in real time to be proactive. Use account equity (balance+open P/L) rather than balance, since balance only updates after closing trades.

- **Preemptive Halts:** As mentioned, if approaching limits, do something before crossing. For instance, at 4% daily loss, maybe disallow new trades or reduce position sizes (cool off). Or if one trade's potential loss could push us over 5%, block it. This is analogous to how firms operate – they often stop trading for the day if losses hit a certain threshold lower than the absolute max.

- **Test Risk Scenarios:** Simulate worst-cases to ensure your risk logic works. For example, feed a sequence where all trades lose – does it stop at correct point?. Simulate multiple strategies piling on exposure – does the exposure check catch it? Also test recovery: if a kill-switch triggered, does the system correctly resume next day or after reset conditions? Possibly require manual reset to avoid auto-resuming without oversight (in FTMO, once you hit max loss, challenge is failed, so you might actually want to fully stop trading until user intervention).

- **Compliance and Audit:** Keep detailed logs of risk decisions. Also, implement versioning as part of compliance: record what version of code or strategy parameters were in use (so if something goes wrong, you can trace back) [17] [18]. Even though that's not directly risk engine's job, it's part of overall risk management culture (we saw references to version registry [17]). Risk Engine might enforce that if code version is not the approved one, don't trade (some advanced setups do a checksum of strategy code and require it matches a known approved hash).

MT5 Integration Points: Risk Engine calls `mt5.account_info()` to get balance, equity, margin etc., possibly on a schedule (e.g., every tick or every few seconds). It may also use `mt5.positions_get()` to see current open positions and calculate combined exposure. However, often risk can maintain that info itself by updating as Execution confirms orders. Still, cross-checking with MT5's view is a good idea periodically in case something desynced. If the account supports equity stop-out (broker will automatically close positions at certain margin level), that's last resort; the system should aim to close well before that. In some cases, one can set broker-side limits (like some brokers allow setting a stop-out equity level or using tools like trading server APIs); but in MT5, not much is built-in beyond stop-loss on trades.

Recommended Prompt Templates for Risk Engine:

- *Code Generation & Refinement:* **"Write a class `RiskManager` with methods: `check_order(order, account_info, positions)` returning True/False or adjusted order if it violates risk (like reducing volume), and `update_after_trade(trade_result, account_info)` to update internal stats. Include daily loss and total drawdown checks."** – This could yield a skeleton. You might refine: **"Add to RiskManager: if daily loss > 5% or total drawdown > 10%, it should trigger a kill_switch. Implement a method `should_halt_trading()` that returns True if those conditions met."** After code generation, review carefully since these calculations are tricky. Possibly ask Claude to explain the output code to ensure it did it correctly.

- *Debugging & Testing:* If something slipped, e.g., **"We noticed the system exceeded the 5% daily loss without stopping. Here's our RiskManager logic… (provide code). Why might it have failed?"** – The AI might spot that you used balance instead of equity, or you didn't count open trades' unrealized loss. Or maybe time-of-day reset issue. This can help pinpoint logic bugs. Another: **"Risk engine blocked a trade erroneously saying exposure limit hit. Provide potential reasons for false positives in exposure calculation."** – Possibly it didn't account that an opposite position reduces net exposure, etc.

- *Updating & Extending:* **"Implement a correlation limit: if two symbols are highly correlated (like EURUSD and GBPUSD), the combined risk shouldn't exceed X. How can I integrate this into risk checks?"** – The assistant may suggest grouping instruments or a correlation matrix to enforce limits. Or, **"Allow an override: sometimes we want to allow one extra trade beyond normal rules if manually approved. How to design a manual override in Risk Engine safely?"** – It might propose a flag set via GUI that temporarily raises a limit, with a clear timeout or scope so it doesn't remain open.

- *Performance Optimization:* Risk computations are usually light, but if many positions and high tick frequency, ensure not to do heavy loops unnecessarily. You can prompt: **"Is my approach of recalculating total P/L on every tick efficient? I loop through all positions for each tick."** – The AI might confirm it's fine if positions count is small, or suggest maintaining a running sum updated by changes to avoid iterating when unnecessary.

By using Claude or Copilot to simulate scenarios (you can even feed an array of P/L outcomes and ask, "what will my logic do?"), you gain confidence in the risk code. Always double-check AI suggestions with actual calculations on a spreadsheet or small test to be sure, given the critical nature of risk logic.

Module: GUI & Dashboard (Real-Time Monitoring and Control)

Purpose & Role: The GUI/Dashboard module provides a **human-friendly interface** to observe the trading system's behavior in real time and to allow limited manual control or overrides. While the trading runs autonomously, an institutional-grade system offers visibility into its operations – akin to a cockpit dashboard for a pilot. The GUI displays key information: current account balance/equity, open positions, profit & loss, risk metrics (drawdown %, margin usage), and possibly charts of market data with strategy indicators. It also provides controls for an operator or risk manager to intervene if needed: for example, buttons to pause trading, kill all positions (panic button), or toggle specific strategies on/off. The GUI is crucial for **situational awareness** and quick reaction – especially in a prop trading context, one might want to manually close a trade or halt the system if something looks off.

Internal Structure: The GUI can be built as a lightweight web app or desktop app. The recommendation given is to use **Streamlit** for a simple web dashboard in Python. Streamlit allows building a dashboard that updates continuously with new data. The module might run as a separate process that periodically queries the core system for the latest status (or subscribes to events). The main components of the GUI:

- **Data Display Components:** These include tables for open positions (listing symbol, size, entry price, current price, unrealized P/L), and recent trades (with timestamps, profit, etc.). Also, text or numeric displays for metrics like *Today's P/L*, *Current Drawdown*, *Available Margin*, etc.. Visual elements could be charts: an equity curve graph (to see performance over time), or live price charts with markers

where trades occurred (to visualize strategy behavior). With Streamlit or Plotly you can embed such charts fairly easily.

- **Control Widgets:** Such as:

- A master **Start/Stop toggle** for trading (when switched off, Strategy Engine should not execute new trades).
- A **Kill Switch button** that when pressed will trigger the risk engine's kill-switch (flatten positions).
- Strategy toggles: list of strategies with on/off switches to enable or disable each strategy module in real time (the Strategy Engine should check those flags before acting on signals).
- Manual trade input: an interface to submit a manual order (like choose symbol, volume, buy/sell, maybe market or pending) – useful if the operator wants to intervene or test something. If provided, this should route through the same Execution Engine (with risk checks) for consistency.

- Parameter adjusters: possibly sliders or inputs for certain parameters (like risk % per trade, or a threshold) if you want to allow live tuning. This can be dangerous if misused but can be handy in a closed testing environment.

- **Alert Indicators:** The GUI can highlight warnings or alerts, e.g., drawdown approaching limit – show a red indicator or pop-up ("Warning: 4% daily loss"). If a kill-switch triggers, display big red "TRADING HALTED – Limit Reached" so user knows. Also could beep or send notification for critical events.

Data Flow & Integration: The GUI should ideally **not slow down** or block the trading logic. A common approach is to have the GUI query the system state on an interval (say every second or few seconds) rather than the system pushing to the GUI synchronously. In Streamlit, you might run it as a separate web server; it can call functions or read status from a shared source (like a database or a memory cache or an API exposed by the core). Another approach: run the GUI in the same process thread with periodic updates – but that can interfere with trading timing if not careful. The recommendation to run GUI as a separate process or thread and communicate via a thread-safe mechanism or network API is wise. For instance, the trading system could maintain a status dict (in Redis or just a local singleton protected by locks), which the GUI reads. Or the GUI could make HTTP requests to a small REST API served by the trading engine that returns JSON status.

Interdependencies: GUI ties with **every module** to present info: from Data Ingestion (prices for charts), from Strategy (which strategies active, any signals?), from Execution (open orders/trades), from Risk (current risk metrics), from Pattern (perhaps showing performance analytics). It also feeds into modules via controls: toggling strategies might call a Strategy Engine method; the kill button triggers Risk/Execution; manual trade goes to Execution. It's important the GUI only allows what's safe – e.g., if manual trades are allowed, still pass them through risk checks. Also, implement some auth if needed (for multi-user or remote access) – but if just local, a simple use is fine.

Best Practices:

- **Non-blocking UI:** Ensure the UI updating doesn't interfere with trading. If using Streamlit, its architecture is usually separate so it won't block trading code. If using a custom GUI in the same app (like a Tkinter or PyQt), run it in a separate thread or process. The bottom line: **trading should**

continue even if GUI is slow or crashes. The system should run headless if needed (e.g., if GUI is closed, trading still runs).

- **Clarity and Relevance:** Design the dashboard to highlight critical info. For example, real-time P/L and drawdown should be prominent (maybe a big gauge or number). Strategies status (active/inactive) should be clearly indicated. If something is wrong (error state or disconnected from MT5), put a noticeable alert. Use color cues: green for normal, amber for warning, red for critical. Make layout clean and not overly cluttered (can use tabs or sections for different info).

- **Refresh Rate:** Updating once a second is usually enough for human eyes; high-frequency updates (like 10 times a second) will just hog resources. Streamlit by default can refresh on script rerun; you might use `st.experimental_rerun` on a timer or something. Find a balance: perhaps update positions and P/L every 1 second, charts every few seconds. The data feed in trading might be tick-by-tick, but GUI doesn't need microsecond fidelity.

- **Security:** If the GUI is accessible over a network (even local network), secure it. Streamlit can be simple, but if on a remote server, use a password or network restrictions because you wouldn't want just anyone viewing or pressing buttons (especially kill or manual trade). In institutional setting, typically only authorized people have access, but still good to implement an authentication step if the dashboard is deployed beyond local.

- **Audit & Confirmations:** For dangerous actions (like kill all, or manual large trade), consider a confirmation prompt ("Are you sure? yes/no") to avoid accidental clicks. Also log any manual actions with user and time.

- **Decoupled from Core:** If possible, design so that the GUI reads from logs or a database rather than querying internal variables, because that further decouples it. For example, the system could log all trades to a SQLite or send them to a InfluxDB or just append to a CSV; the GUI could read from there to display history. This way, even if GUI was started later, it can reconstruct history from logs (helpful for monitoring longevity).

MT5 Integration: The GUI itself might not directly call MT5 functions – it should rely on the core for that. However, one might integrate an interactive chart by fetching some data via MT5 (but easier to get it from the data module's storage). If using an existing solution like MQL5's built-in web GUI (rarely used) or some bridging, not needed. Possibly, the GUI might also interface with other resources: e.g., display an economic calendar (via some API like ForexFactory RSS) on the side; that could be nice to have in the dashboard.

Recommended Prompt Templates for GUI & Dashboard:

- *Code Generation & UI Layout:* **"Use Streamlit to create a trading dashboard. It should display: account balance and equity, open positions in a table (symbol, size, P/L), and have buttons to pause trading and close all positions. Show an example of updating these in real-time."** – This prompt will likely produce a simple Streamlit app code. You can refine: **"Add a plot of the equity curve. Also include checkboxes to enable/disable strategies named StrategyA, StrategyB."** – The assistant might show using `st.line_chart` or matplotlib for equity, and `st.checkbox` for toggles. Ensure to integrate those toggles with something (maybe storing their state and printing it, you'd later connect to strategy control logic).

- *Debugging & Synchronization:* **"Our GUI sometimes shows stale data (e.g., a position that was closed still appears for a few seconds). How can we ensure the dashboard always shows the latest state? What techniques in Streamlit or web apps can help reduce latency or race conditions?"** – The AI might suggest using Streamlit's caching carefully or using WebSocket for pushing updates (though Streamlit doesn't natively push without rerun). Possibly it might mention adjusting the refresh interval or double-checking that the core is updating the source of truth in time.

- *User Interaction Handlers:* **"Write an example of how the GUI's 'Pause Trading' toggle can communicate with the main trading loop. For instance, if `pause_flag` is True, skip sending new orders. Suggest an implementation."** – The assistant might say: have a global or context object that the trading loop checks, and in Streamlit when toggle is clicked, update that object. It may discuss using something like `st.session_state` or a common file/flag. You might refine by asking about thread-safety if using multi-threading.

- *Extending Features:* **"How to integrate a live price chart in the dashboard? The GUI currently plots equity, but I also want to see a candlestick chart of EURUSD with markers where trades happened. Provide suggestions or libraries."** – Possibly it will mention Plotly's `plotly_chart` for candlesticks or Streamlit components.

- *UI/UX Improvement:* Copilot might help with layout: as you type Streamlit code, it can suggest how to arrange columns or tabs. For example, writing `col1, col2 = st.columns(2)` and then filling in might yield nice suggestions.

Remember, a polished GUI often requires iterative tweaking – you can use the AI to generate the initial code, then actually run it and see how it looks, then adjust prompts for improvements. Because the user experience is subjective, you might ask the AI open-ended questions: **"What are the most important elements to show on a trading bot dashboard to ensure compliance with FTMO rules and strategy monitoring?"** – It may list points you can check against your design.

Module: Macro Synchronization Layer (Economic Events Integration)

Purpose & Role: The Macro Synchronization layer ensures that the trading system remains **aware of major external events and regime changes** in the market (especially macroeconomic news) and synchronizes its activity accordingly. Markets can become extremely volatile or behave unpredictably around events like central bank rate decisions, employment reports, or geopolitical news. A common professional practice is to avoid trading during these times or switch to a specialized strategy that handles volatility. Thus, this module's role is to **shield the system from known risks and exploit known opportunities** in the macro calendar. It does so by (1) tracking a calendar of scheduled news events and implementing pre/post event behavior, and (2) detecting unscheduled shocks via real-time indicators (like sudden volatility spikes or news sentiment feeds) to trigger protective measures.

Internal Structure & Responsibilities:

- **Economic Calendar Integration:** The module has access to a feed of scheduled economic releases (e.g., from an API like ForexFactory, Investing.com, or even a manually maintained calendar file). It parses upcoming events with their time, importance, and currency impacted. For instance, FOMC interest rate decision at 14:00 NY time affecting USD, or Non-Farm Payrolls (NFP) first Friday of month, etc. The module likely runs a scheduler or loop that constantly checks current time against event times. When a significant event is, say, 5 minutes away, it issues a **"pre-event" signal** to the rest of the system to prepare. That could mean: stop opening new trades (freeze signals) and perhaps tighten stops on open positions or close them entirely if you have a policy not to carry risk into the number. Then, after the event, it can either automatically resume trading after X minutes or wait for volatility to normalize by some criteria. The module may also label the period around an event so that performance during those times can be analyzed separately (e.g., to see if strategy tends to lose around news).

- **Real-Time News/Sentiment Feed:** For unscheduled news (e.g., surprise announcements, tweets by officials, etc.), a full AI-driven sentiment feed (like RavenPack, Reuters News Analytics) might be overkill here, but at least a simpler approach: monitor volatility indices or price movements as proxies. For example, if a typically quiet currency pair suddenly spikes 50 pips in a minute outside known events, assume an unscheduled news and trigger a similar pause. If one has access to a news headlines feed (some are free, some paid), the module could scan for certain keywords ("unscheduled rate cut", "war", etc.) and alert the system. But likely, monitoring market data for anomalies is straightforward: e.g., if 1-minute ATR or spread jumps beyond a threshold, call an internal **"volatility surge" event**. The Risk Engine could then treat that like a macro event and either freeze or reduce risk.

- **Coordinating Strategy Behavior:** Macro Sync doesn't directly execute trades; rather it **controls the state of strategies/execution**. Implementation could be: Macro module sends events/signals to which strategies/risk respond. For example, it might broadcast an `Event: TRADING_PAUSE start 13:55 for EUR/USD (FOMC)` – strategies subscribed would automatically not trade EUR/USD at that time. Or Macro can directly set a global flag that Strategy Engine checks (like `if macro_pause: skip signals`). It could also instruct Execution to enforce no trades or widen slippage tolerances if needed. In some systems, a specialized "NewsTradingStrategy" might turn on around events if you have one (but often, the approach is to stay out). Macro Sync might also manage **session-based logic**: e.g., no trades during illiquid hours or particular day-of-week quirks (some consider Monday morning or Friday afternoon as special regimes – not exactly macro, but related concept of regime).

- **Resumption & Adaptation:** After an event passes, Macro Sync should determine when to resume normal trading. A simple approach: wait N minutes after the event time. A smarter approach: watch volatility and spreads, resume only when they fall back to normal range. This can be done by monitoring, say, the ATR or spread; once spread is back to typical levels and price movements stabilize, send a "resume trading" event. This avoids the case where an event triggers huge volatility and even after the scheduled time, the market is still chaotic for a while.

Inputs & Outputs: Inputs include an event calendar (could be read from a file or API at startup or daily) and potentially a live news or volatility feed. Another input might be manual – e.g., a user might manually input

an ad-hoc event (like “Fed Chair speaks at 3pm – pause trading then”) if not in the official calendar. Outputs are internal signals/events to other modules: e.g., **PauseTrading**, **ResumeTrading**, or **VolatilityAlert** events. It may also output logs for every event (for audit: “Paused trading 5m before ECB rate decision”). Possibly it outputs info to GUI (like a countdown or display “No trading: NFP release in 2min”).

Interdependencies: With **Strategy Engine**: strategies must obey macro instructions; either Macro directly disables strategies or Strategy checks Macro’s status. With **Risk Engine**: macro events may cause risk changes – e.g., risk might reduce allowed leverage ahead of an event. Risk also listens to volatility alerts – e.g., if Macro triggers “volatility freeze”, risk might decide to close open trades if they exceed a risk threshold, effectively acting as an emergency stop. With **Execution**: to enforce a pause, Execution might reject any orders that somehow come during a forbidden interval (extra safety). Also if a news event requires closing positions (some firms flatten positions before big events), Macro can instruct Execution to do so. With **GUI**: show upcoming events and allow user to override (like maybe skip a planned pause if the user wants to trade news, though for FTMO usually you avoid risking it).

Best Practices:

- **Reliable Event Feed:** Use a reliable and accurate source for the economic calendar. Ensure times are correct and time zones aligned. If the trading server is in GMT and calendar in EST, convert properly. Also consider Daylight Savings. Many API providers give times in UTC or absolute epoch which helps. The system should ideally load the day’s events at start and keep them in a schedule.

- **Categorize Events by Impact:** Not all news needs a trading halt. E.g., Nonfarm Payrolls or Fed decisions are high impact (red events) and you pause trading. Medium impact (orange events) maybe you just issue a warning or slight caution (maybe reduce size), and low impact (yellow) ignore. The module can have a filter: only act on events flagged high-impact for instruments you trade. This avoids unnecessary interruptions.

- **Testing the Pause Logic:** You wouldn’t want the system failing to resume after a pause. Test that after an event time passes, strategies indeed resume. Also test overlapping events (what if two events close by? E.g., Canadian and US jobs both at same time – ensure handling multiple). And what if an event triggers while already in a macro pause from another event – handle gracefully.

- **Volatility Cutoffs:** Determine sensible thresholds for “market shock” detection. Could be based on instrument ATRs or known baseline volatility. Possibly maintain a rolling measure and if the latest movement is e.g. 5 standard deviations beyond normal, call it a shock. Also measure spreads: if spread widens drastically (like during Swiss franc unpeg, spreads blew out), that’s a sign too. The Risk Engine might also directly monitor spread and trigger kill-switch if spread beyond some pips (because wide spread can hit stops unexpectedly or cause margin issues).

- **Special Strategies Around News:** If you ever decide to trade news (some strategies do straddle orders around events), keep it separate and controlled. Likely out of scope here, but ensure that a general pause doesn’t accidentally stop a strategy meant for news. That said, in FTMO context, probably safer to not trade unknown news.

MT5 Integration Points: MT5 itself doesn’t provide news feeds to the Python API (it has a news tab in terminal, but not exposed to Python as far as I know). So the macro module will use external data (like an

HTTP request to a calendar API). Could schedule via Python's `schedule` library or simply loop sleeping until event times. It will use the system clock for scheduling – ensure that's synced and align with broker time if needed. Another integration: checking spread – can get via `mt5.symbol_info_tick` which has bid/ask, compute spread difference.

Recommended Prompt Templates for Macro Sync:

- *Code Generation & Data Handling:* **"Generate Python pseudocode to load an economic calendar (with datetime and impact) and implement a loop that checks if any event is within 5 minutes. If so, print a message or trigger a function to pause trading, and then resume 15 minutes after the event."** – This prompt yields the structure of scheduling. Then, **"How to integrate this with the trading system? For example, how can I signal the Strategy Engine to pause? Suggest a mechanism."** – Likely suggestions: global flags, or sending a message on the event bus that Strategy Engine listens for. The assistant might mention specific libraries (like `schedule` or `apscheduler`) to manage timing which could be helpful.

- *Debugging & Edge Cases:* **"We had an event at 15:00 where trading was supposed to pause, but a trade still happened at 14:59:30 and got caught in volatility. How to adjust the system to avoid this? (e.g., pause slightly earlier or close out positions)"** – The AI may suggest increasing the lead time or explicitly flattening positions before. Another: **"The macro module missed resuming trading after an event because volatility stayed high for 30 minutes. Meanwhile, some signals were skipped. How to handle prolonged volatility? Should we have a maximum pause duration or manual intervention alert?"** – It could propose a cutoff where after, say, an hour, it alerts that system still paused and requires input.

- *Updating & Extending:* **"Add unscheduled news detection using volatility: if 1-minute price change exceeds X or spread > Y, invoke an immediate trading halt (and maybe close positions). Provide a strategy for this."** – The AI might suggest maintaining a rolling window of price changes and using threshold. Or using a volatility index (like VIX for equities, but for currencies one might use OIS spreads or something – likely too advanced, stick to price-based detection). Implementing immediate pause is straightforward: risk can catch it, or macro itself can just call kill-switch if movement beyond tolerance (though that's drastic – maybe better to just pause new entries and tighten stops). The assistant can outline the logic.

- *Integration Test via AI:* Possibly after coding, you can describe a scenario and ask the AI what should happen: **"Scenario: It's 5 minutes to Nonfarm Payrolls, we have 2 trades open. What does Macro Sync do? Now at 2 minutes to NFP, a new signal appears – what happens? News hits, volatility spikes 3x normal, what actions are taken and when do we resume trading?"** – This is like a narrative test; the AI's answer can confirm if our intended workflow is correct or if we missed something.

Working with AI on this module helps because timing and external data can be tricky – it might help to ensure no corner cases are missed (like overlapping events or daylight savings). Always cross-check the logic with actual market event timing to ensure it aligns with how FTMO's clock works (they often reference times in the user's local time or CET for their own tracking).

Module: Pattern Detection & Machine Learning Feedback (Continuous Improvement)

Purpose & Role: The Pattern Detection & ML Feedback module serves as the **self-analytical component** of GENESIS – it monitors the system's performance over time and uses data-driven techniques (from basic statistics to machine learning) to **identify patterns and biases** in trading outcomes and then feed improvements back into the strategy [19]. In essence, while other modules handle the day-to-day trading, this module works in the background (or off-hours) to answer questions like: "Which market conditions favor our strategies, and which hurt?"; "Are our trade signals losing effectiveness?"; "Can we tweak our parameters to adapt to recent market changes?"; and "What common factors characterize our losing trades that we might avoid in future?". By continually learning from the trading history, the module can suggest or automatically implement adjustments – making GENESIS a system that improves over time rather than stagnating.

Internal Structure & Responsibilities:

- **Data Collection:** This sub-component aggregates **detailed logs of trades and signals**. After each trade (or even each signal whether traded or not), relevant data is recorded: the features at the time (indicator values, volatility, time of day, etc.), what action was taken (or not taken), and the outcome (profit/loss, max adverse excursion, etc.). The data could be stored in a database or simply a CSV that grows. It's important to include context so patterns can be found (like tag each trade with strategy ID, market regime indicators, and outcome). This log serves as the training dataset for analysis.

- **Performance Analytics:** On a continuous or periodic basis, the module crunches the historical data to produce metrics: Sharpe ratio per strategy, win rate, average win/lose, max drawdown, etc. But deeper, it slices the data: e.g., performance by hour of day, by day of week, by volatility level, by trade hold time, etc.. For instance, it might find Strategy A is profitable in trending markets but loses in ranging ones – which is a pattern to address. Or maybe all strategies lose on Mondays – maybe due to weekend gaps or low liquidity. These insights can be surfaced to developers or trigger automated rules (like "don't trade Mondays" if clearly bad).

- **Bias Detection:** The module looks for bias or drift in the strategies. Are we overfitting to one type of market? Is a strategy's edge decaying? For example, maybe an MA crossover used to work but now every crossover trade loses (meaning the signal's Information Coefficient has become negative) – that could indicate alpha decay. The module could flag this and recommend turning off that strategy or retraining its model. It also might catch operational issues – e.g., consistently high slippage on a particular broker or time, indicating execution bias (like we get bad fills during Sydney session – maybe liquidity issues).

- **Automated Tuning (Auto-ML):** Perhaps the most advanced part: using the collected data to **adjust strategy parameters or models**. One approach is **walk-forward optimization** where periodically (say weekly or monthly) the system re-optimizes certain parameters on recent data. For example, it might fine-tune the length of a moving average or the threshold of an indicator to what would have worked best in the last month, then use that going forward (assuming market regimes evolve). Done carefully, this can keep strategies from becoming stale. Another approach is training a **meta-model** (as described in research) that acts as a secondary filter on signals. For instance, train a classifier that

given a signal and context predicts probability of a profitable outcome; use it to filter out likely losers. This is known as meta-labeling and has been shown to improve precision. The module could retrain such a model every so often using recent trades as training examples (label trades win vs loss, learn patterns leading to wins).

- **Pattern Alerts & Updates:** The module can operate in recommendation mode or auto mode. In recommendation mode, it might output messages: "Strategy B's Sharpe in last 50 trades is -0.5, consider disabling it" or "Trailing stop of 50 pips would have saved X amount last week, consider tightening stops." In auto mode, it might actually implement changes: e.g., modify a config file to change that trailing stop from 100 to 50 pips, or set a strategy's active flag to off if performance < threshold. For safety, initially such changes might require human approval (via prompt engineer or config review) because automated strategy changes can be risky if not well-monitored.

Inputs & Outputs: Inputs are the trade and performance data (from logs, backtester results, etc.), as well as possibly external data like new technical indicators or alternative data for finding correlations (though likely we stick to internal data). Outputs are **recommendations or adjustments**. For example: - Parameter tweaks (like "MA length = 20 might be better than 50 for Strategy C given recent vol"). - Strategy enable/disable decisions (maybe output a list of strategies ranked by recent performance). - Updated ML models (the meta-model could be output as a pickled model that the Signal Intelligence uses to filter signals). - It could also feed into the GUI: showing performance trends, showing which strategies are contributing most or least (like a leaderboard of strategies). Also, if an auto-tuning happened, log it (e.g., "Autotuned StrategyA parameter X from 0.8 to 0.7 based on last 100 trades").

Interdependencies: This module depends on **Logging** (Compliance Logging module must provide detailed logs). It may integrate with the **Backtester**: e.g., to test a new parameter, it might run a quick backtest on recent historical data to verify improvement before applying (if you have the capability to simulate quickly). With **Signal Intelligence**: if a meta-model is used to filter signals, that meta-model is built here and then deployed into Signal Intelligence for real-time use. With **Strategy Engine**: if deciding to turn off a poor strategy or adjust risk per strategy, it communicates to Strategy Engine to implement that. Possibly with **Execution**: analyzing execution quality (slippage, fill rates) can yield changes like "switch to limit orders more often" if slippage is consistently high – such suggestion would alter Execution tactics (maybe manual adjustment by developer or could directly adjust a config that execution uses for order type preferences).

Best Practices:

- **Don't Overfit Updates:** It's easy to chase the latest performance and damage future performance (overfitting to noise). So any auto-adjustment should be made with caution and with statistical significance. E.g., require enough trades or a clear pattern before acting. If a model is retrained, use techniques like cross-validation on past data to ensure robustness. Or, if possible, paper-test new parameters in simulation (shadow mode) before fully deploying.

- **Gradual Learning:** Some systems employ a *learning rate* concept for strategy tweaks – e.g., if a strategy's performance is down, maybe reduce its trade size rather than kill it immediately (because it might come back). So Pattern module could suggest risk scaling: like "Strategy B is not doing well, temporarily cut its risk by 50% until it shows recovery" instead of outright off. This avoids frequent on/off flipping. Similarly, parameter changes could be moderate rather than drastic leaps.

- **Track Changes:** Keep a history of what changes were made when (akin to experiment tracking). So if something goes wrong after a change, you can revert. The compliance logging should note "At time X, parameter Y changed from A to B by auto-tune" for audit.

- **Leverage Backtesting**: When the Pattern module finds a promising alternative parameter or filter, ideally run a quick backtest on a decent dataset to verify it improves things (the module can call the Backtester programmatically with the new param). Only then apply to live. This is an advanced integration but very powerful for confidence.

- **Human Oversight:** In institutional contexts, even if such automated learning exists, typically human quant researchers would monitor it. For our context, since we have prompt engineers and Copilot, maybe the pattern module outputs its findings to a file or report that a developer (with AI assistant) can review and then decide to implement via a prompt. This might actually align more with how the user might use this prompt book – perhaps using AI to analyze logs is part of the workflow.

MT5 Integration Points: Not direct – the analysis is offline with the data. It might request historical price data if needed to analyze pattern in market data vs trades. But likely it uses the internal log of trades which already contain needed data. If needed, could use MT5's history (mt5.history_deals_get) to double-check trade outcomes or retrieve older trades that predate our logging.

Recommended Prompt Templates for Pattern/ML Feedback:

- *Data Analysis with AI:* Actually, one can enlist Claude 4 to do some of this analysis on demand. For instance, after a month of trading, you could feed a summary of trades to Claude and ask for patterns. But building it into the system is more formal. For code: **"Provide a Python outline for analyzing trade logs: calculate win rate, average P/L, and separate results by market volatility regime. Use that to identify if strategy performs differently in high vs low volatility."** – The AI may suggest loading data into pandas, adding a column for volatility regime (maybe precomputed or using historical price moves), then groupby to compare outcomes.

- *Meta-Model suggestion:* **"How can we implement a meta-labeling model to filter signals? Describe steps to create training labels from trade outcomes and features to use (like recent signal win/loss, volatility, time)."** – The AI might recall the Lopez de Prado method: use triple barrier labeling, train a classifier. It could suggest a simple approach: label each trade if it was win or loss beyond a threshold, then train logistic regression. As code: **"Show an example of using scikit-learn to train a logistic regression that predicts trade outcome using features like RSI, volatility, and time of day from a dataset of past trades."**

- *Tuning Example:* **"We suspect our moving average period might be suboptimal. Suggest a procedure (and code) to evaluate performance of different MA lengths on historical data and pick the best."** – It may outline doing backtest runs for each parameter or using grid search, then picking the highest Sharpe or return. Possibly recommending out-of-sample validation.

- *Continuous Learning Setup:* **"What's a safe way to periodically update strategy parameters? For example, update every month using last 3 months of data, but ensure not to react to random short-term noise."** – The assistant might mention requiring a minimum number of trades, using statistical tests to see if performance change is significant, or limiting magnitude of changes.

- *Using Copilot for Patterns:* The user (with prompt engineering) could also manually do some of this: e.g., export trade log to CSV, then ask Copilot or use Python to generate pivot tables or charts. The prompt book can mention that synergy: that this module's output can be interpreted by the developer with AI assistance. But since we want an autonomous angle too, we focus on the module.

The goal is that over time, GENESIS doesn't remain static. It learns from its own data similarly to how a quant team would refine a strategy. It's ambitious to fully automate, but even partial steps (like highlighting issues) are valuable. Using the AI tools, one can even have a conversation with the system's data: e.g., feed trade log to Claude with 100k context to get deep analysis. That might be out-of-band usage rather than built-in though.

Module: Backtester (Simulation & Strategy Validation)

Purpose & Role: The Backtester module provides an **environment to simulate GENESIS's trading strategies on historical data** to evaluate performance, test changes safely, and ensure that strategies behave as expected outside of live trading. It's essentially a sandbox that mimics the live system but using past market data (or a data stream) as input. For a complex system like GENESIS, a robust backtester is key to validating improvements from the Pattern/ML module, trying out new strategies, and ensuring that any modifications (like new risk rules or execution tactics) do not inadvertently harm performance or cause errors. It's also used to **replay historical scenarios** (like volatile events) to see how the system would have reacted, which is crucial for debugging and refining the algorithms.

Internal Structure & Responsibilities:

- **Historical Data Feed:** The backtester needs access to historical price data for the instruments traded (tick data or at least bar data). It will feed this data into the Data Ingestion module or a simulation thereof. Often, this is implemented by reading historical ticks/bars from files or a database and pushing them through the event bus just like live data. That way, the Strategy Engine and others don't know the difference – they react as if it were live. The backtester might allow variable playback speed (fast-forward through days of data quickly) or step-by-step for debugging.

- **Simulated Execution:** When strategies generate orders in backtest, instead of sending to real MT5, the backtester simulates order execution. This includes matching against historical price data: e.g., if a strategy placed a buy at limit price X, the backtester will check if the historical price touched X (and when) to determine if it would have filled, partially or fully. It must also simulate slippage and spreads: a simple approach is to fill at the next available price or use the bar's high/low and assume worst-case within that (some backtesters assume if price went beyond limit, you got filled at your limit; for market orders, perhaps execute at bar open+some slippage). The simulation should incorporate **trading rules** like stop-loss and take-profit triggers (if a position is open, and historical price hits the stop, close it at that price). Essentially, it acts as a fake broker.

- **Strategy Harness:** We might integrate an existing framework like Backtrader or LEAN, but given GENESIS's custom architecture, perhaps we implement a light harness that connects the pieces. For example, one could subclass or reuse GENESIS modules but with modifications: a DataFeed that reads from history, an Execution that doesn't call MT5 but instead logs trades and updates positions in a simulated portfolio.

- **Metrics & Reporting:** After (or during) a backtest run, the module computes performance metrics: total return, max drawdown, daily P/L distribution, Sharpe, etc. Also, it can produce a detailed trade ledger for analysis. This output can be fed to the Pattern Detection module for further analysis or simply to the developer. If integrated, one could run the backtest automatically on recent data each day to see how changes would have done, or run it for many parameter variations.

- **UI Integration:** Possibly integrate with the GUI: maybe have a mode to run a backtest from the dashboard (like specify a date range and run). But more likely this is used offline by devs.

Inputs & Outputs: Inputs: historical price data (maybe as Pandas DataFrames or via API from a data provider), initial account settings (starting balance, etc.), perhaps a configuration of which strategies to test and their parameters. If we want to test prospective changes, we input those changed parameters or code (so often you re-run backtest after code modifications in development). Outputs: performance statistics and logs of simulated trades. In some systems, you might feed those logs into the Pattern module to update the ML model (like using backtest data to train meta-model, though careful to avoid lookahead bias – usually training should only use past data relative to a point in time).

Interdependencies: With **System Architecture:** to maximize parity, the backtest environment tries to reuse the same Strategy logic and Risk logic as live. It might bypass or simplify some parts (like GUI, obviously not needed, and possibly simplify risk if some things like kill-switch triggers you may or may not simulate). But ideally, one codebase runs both by abstracting live vs backtest differences (e.g., using an interface for data feed and execution that can have a "live" implementation (MT5) or a "simulated" implementation). The **Pattern Detection** can use backtester to test alternative strategies: e.g., try adding a filter and see if performance improves on last year's data. The **Risk Engine** can also be tested via backtest to see if it would have prevented historical drawdowns properly.

We saw references to Backtrader and event-driven backtesting in the case study. Possibly this module could literally integrate Backtrader library, which already supports running a strategy on historical data with broker simulation. But integrating our custom code into Backtrader might be non-trivial. Another approach: code our own simple simulator or use a simpler open-source like `backtesting.py` or others.

Best Practices:

- **High Fidelity Simulation:** The closer the backtest to reality, the more reliable its results. If possible, use tick-level data for strategies that work on tick level, to capture intra-bar price movement, which affects fill accuracy and stops. If only bar data, consider assumptions (like using OHLC – some use "next bar open" execution which can be unrealistic if strategy triggers mid-bar). If strategies are slower (e.g., operate on M15 or H1 bars), bar data is fine. For fast strategies, invest in tick data.

- **Include Costs:** Simulate spreads, commissions, slippage. For FTMO, commission is typically zero but spread exists and perhaps some overnight fees. At least subtract a spread on each trade entry/exit in simulation. Slippage can be modeled as a random or worst-case within a bar (e.g., for market orders, maybe slip half the spread or more if volatile).

- **Validate with Live:** Do occasional **forward-testing** (paper trading) to ensure backtest matches live. This means running the system in demo in parallel and comparing outcomes to backtest for the

same period – any large discrepancy indicates backtest may not be accounting for something (like maybe in live some orders didn't fill that backtest assumed, etc.).

- **Speed and Efficiency:** Backtesting large histories can be slow, especially tick-by-tick in Python. If needed, optimize: vectorize parts (though event-driven logic is inherently stepwise), or restrict to key periods for analysis. Possibly allow multi-processing to test multiple parameter sets concurrently.

- **Scenario Testing:** Use the backtester for stress scenarios: feed it data from known market crises (2008, 2015 CHF, 2020 Covid crash) to see how the system would have fared. This might reveal issues (like risk module might not handle gap well because our simulation might not either, etc.).

MT5 Integration Points: Interestingly, MT5 itself has a Strategy Tester in the terminal (for EAs), but for Python we can't directly use that. However, one could conceive running an EA that calls the Python strategy as signals, but that's complicated. So probably no direct MT5 usage here; we rely on offline data. Possibly use `MetaTrader5.copy_rates_range` to fetch historical bars for backtesting if not already have data, but doing that repeatedly is inefficient. Better to once fetch and store locally.

Recommended Prompt Templates for Backtester:

- *Code Design:* **"Design a simple backtesting loop for GENESIS: iterate through historical price data (timestamp, open, high, low, close), feed each price tick to the strategy logic, capture any orders, and simulate execution of those orders on the data. Outline how to structure this."** – The AI may outline reading a CSV of prices, a for-loop where at each step it calls strategy.generate_signals(), then executes them, etc. Could mention using a data structure to hold open positions.

- *Utilizing frameworks:* **"Would using an existing backtesting framework like Backtrader or Zipline be beneficial here? How could we integrate GENESIS strategy code into such a framework?"** – It might weigh pros/cons. Possibly mention that Backtrader could run our strategy if we wrap it as a Backtrader strategy, and it has support for custom observers, analyzers (like risk metrics).

- *Debugging: If a backtest result seems off: "Our backtest shows much better results than live trading. List possible reasons for discrepancy (lookahead bias, data issues, execution assumptions) and how to mitigate them."** – The assistant will likely enumerate common pitfalls: using future data accidentally, ignoring realistic slippage, differences in data quality, etc. For each, address by adjusting code or method.

- *Speed concerns:* **"Backtesting one year of tick data is slow. What strategies can improve performance? (e.g., using numpy/pandas for vectorized backtest, simplifying model detail, or filtering ticks)"** – Could mention not simulating every tick if strategy doesn't need it (like only process relevant ticks or bars), or using PyPy/Numba to accelerate Python loops, or distributed computing to test multiple things in parallel.

- *Integration with AI Tools:* It's less about prompting AI during backtest run, but you might use AI to analyze backtest results. For example, after running, you could ask Claude: "Here is the trade-by-trade result of a backtest, what patterns do you see?". That's more pattern module usage though.

The prompt book might mention that a developer can use such analysis with AI as part of their workflow.

Using Copilot, one could also get a lot of the boilerplate for backtesting, as it has seen similar code. But making sure it aligns with our actual system's structure is key.

Module: Compliance & Audit Logging (Transparent Record-Keeping)

Purpose & Role: The Compliance and Audit Logging module is responsible for **recording all actions and decisions** of the GENESIS trading system in a tamper-evident, structured manner to ensure that everything is auditable after the fact. This serves multiple purposes: it helps meet any regulatory or prop firm requirements for record-keeping, it provides a way to debug and trace what happened in complex scenarios, and it gives confidence that the system adheres to the rules (e.g., if FTMO asks why a rule was or wasn't violated, you have the data). Essentially, this module is the "black box recorder" of the trading system – logging every order, every risk check, significant events, and any overrides or manual interventions [15] [20].

Internal Structure & Responsibilities:

- **Logging of Trade Activity:** Every order sent to the broker (or simulated execution in backtest) should be logged with details: timestamp, symbol, volume, order type, price, and result (filled, price, or if rejected, the error code) [15]. Also log when positions are closed, either by take-profit/stop-loss or manual. Essentially, one should be able to reconstruct the entire trade history from these logs alone, including the state of positions over time.

- **Logging of Decisions & Signals:** It's useful to log signals and strategy decisions, though perhaps at a slightly less granular level to avoid log overload. For example, log when a strategy triggers a trade (and why: "MA crossover buy signal triggered for EURUSD at 1.1000, volatility low, confidence 0.8"). If a signal was ignored due to a filter or risk, log that too ("Signal at 12:00 ignored by risk management – daily loss limit reached"). Similarly, log any strategy being toggled on/off, or any parameter changes (especially if via ML feedback) [21].

- **Risk & Compliance Logs:** Risk events should be clearly logged: hitting of limits ("Daily loss 5% reached, triggering kill-switch at 14:05:23, closed positions"), any trade blocked ("Order to buy 1.0 lot GBPUSD at 1.30 blocked – would exceed exposure limit"), etc. Also, periodic risk status can be logged (e.g., end of day summary: max DD, etc.). If interacting with external compliance processes (like submitting reports), those can be prepared from logs.

- **System Health & Telemetry Logs:** Possibly integrated here or Telemetry module: things like heartbeat ("System heartbeat: all OK at 13:00:00") [22], or any error conditions ("Lost connection to MT5 at 13:05, reconnecting... success"). This helps detect if the system froze or crashed at some time, by seeing no heartbeats.

- **Versioning and Change Logging:** When the system starts, log the version of code or commit hash and config versions it's running [17] [18]. If any strategy or risk parameter is changed (especially by Pattern/ML module automatically), log that with timestamp and old/new values [21]. This is crucial

for compliance – if an unexpected trade happens, you need to know which code was responsible at that time.

- **Storage of Logs:** The logs should be written to a **persistent storage** (files or database). Plain text rolling log files are common. Using a structured format like JSON or CSV can help parse them later. For audit, you might even have redundant logging: one human-readable log, and one structured event log. The logs should have timestamps in a consistent timezone (ideally UTC) with high resolution (ms if possible) for ordering events precisely.

- **Access Control & Security:** Ensure logs cannot be tampered easily. For extreme cases, you could implement an append-only log or even send to an external server so that even if the trading server is compromised, logs remain. Given scope, likely not needed, but at least protect log files from unauthorized editing. If multiple people or processes have access to the system, log any manual interventions with user identity (like if someone clicked "close trade" in GUI, log which user did it and when) [23] [24] .

Inputs & Outputs: The module doesn't have external inputs (aside from receiving events from all other modules to log). In practice, it might subscribe to the event bus to catch all events and write them to log. Alternatively, other modules call a logging function with relevant info (e.g., Execution calls log when order sent, Risk calls log when limit triggered, etc.). Output is the log repository. Optionally, it can generate summarized compliance reports (like an end-of-day report of all trades and P/L, if needed). If working with a prop firm, sometimes they require you to submit logs or just keep them in case – with FTMO typically you just avoid violations rather than submit logs, but good to have anyway.

Interdependencies: With **all modules**: everything that happens should funnel a message to this logging system. Particularly: - Execution and Risk are heavy log sources (orders and risk events). - Strategy Engine logs signals/trades decided. - Macro logs events triggered (like "Trading paused for news event at X"). - Pattern/ML logs parameter changes or at least recommendations given. - Telemetry logs performance issues (like "Latency spike: 120ms at 12:31"). - GUI logs manual actions (kills, toggles). Integration wise, often you implement logging as a common library that everyone calls, rather than a separate event consumer – but either way is fine.

Best Practices:

- **Structured Logging:** Each log entry at least include a timestamp, module/source, and message. Better, log key fields too (like "ORDER_EXECUTED: id=12345, symbol=EURUSD, side=BUY, volume=0.1, price=1.1045, sl=..., tp=..., P/L=..."). Structured logs facilitate quick searching and analysis (like grep or parsing by scripts).

- **Time Sync:** Use an accurate clock (sync to NTP). If analyzing across multiple systems (like comparing to broker logs), consistent time is crucial.

- **Log Rotation and Retention:** Logs can grow huge (especially if tick events were logged, but we likely don't log every tick, just decisions). Implement rotation (daily logs or size-based). Archive older logs if needed. Ensure disk space won't run out by log accumulation.

- **Testing Logging:** It's dull but verify that critical things are indeed logged. A risk is you assume something is logged but a code path didn't call it. You might create a test mode where system runs for an hour and then check log to ensure at least one heartbeat per minute, etc.

- **Use of Logs for Debugging:** The prompt engineer can use the logs with AI assistance to investigate issues. For example, if a trade violated a rule, you could feed the relevant log lines to GPT to see step-by-step what happened or to identify anomalies. The thorough the logs, the easier this is.

- **Compliance Considerations:** If eventually connecting to a real broker or fund, they might require storing logs for X years. Using CSV or database might ease retrieval. For now, text logs suffice.

MT5 Integration Points: Not direct, except log any data from MT5 events, e.g., `mt5.last_error()` code when something fails, account info snapshots, etc. Possibly also log the account equity each day or trade for record.

Recommended Prompt Templates for Logging:

- *Implementation:* **"Provide a Python snippet for a logging utility that writes events to a file with timestamps. It should be thread-safe (if multiple threads write) and have functions like log_order, log_risk_event, etc., that format messages consistently."** – Might yield a usage of Python's `logging` module or a custom file write with a lock.

- *Log Verification:* **"We want to verify through logs that the system never violated FTMO rules in a test run. What patterns would you search for in the logs? Suggest a strategy to automatically detect any potential rule violations using the logs."** – The AI might say: search for any time where drawdown exceeded threshold, or daily loss beyond X, or maybe confirm that right after a violation threshold the logs show kill-switch triggered (meaning it worked). Could even propose writing a script to parse logs for compliance.

- *Leveraging Logs with AI:* **"In case of a strange behavior (like an unexpected trade), how can I use the logs and possibly an AI assistant to diagnose it?"** – Might answer: filter log around that time, feed the sequence of events to AI to interpret, etc. This highlights how comprehensive logs aid both human and AI understanding.

- *Documentation:* Possibly ask for log file structure doc: **"List the key log entries that should be included for full audit of trading: from initialization to shutdown."** – Ensures we don't forget anything (the AI might mention linking code version, heartbeats, each order, each risk limit trigger, etc. which we covered).

By following these logging practices, any anomaly can be traced and the system's behavior can be defended or explained. For instance, if FTMO says you hit 5.1% daily loss, you can show logs that at 5.0% loss you closed trades, but slippage on closure took it to 5.1% – which might help in an appeal with evidence (this is hypothetical, but shows the value of detail).

High-Level Architecture Diagram & Event Flow

Figure: Modular Architecture & Event Flow. The GENESIS system is composed of loosely coupled modules communicating via an internal event bus. Market data from MT5 flows into the Data Ingestion module, which publishes tick/bar events consumed by the Signal Intelligence and Macro Sync modules. Signal Intelligence produces enriched trade signal events that are fed into the Strategy Engine. The Strategy Engine, considering current positions and risk, issues trade order events. Before execution, the Risk Engine intercepts order events for pre-trade checks (enforcing limits, etc.). Approved orders pass to the Execution Engine, which places them on MT5 and emits execution results (fills, positions) back into the event stream. The GUI subscribes to status events (positions, P/L, risk alerts) for display and can inject control events (like "pause trading" or "close all") that modules heed (e.g., Strategy Engine halts on pause, Execution executes close-all). The Pattern Detection module continuously listens to trade outcomes and performance metrics, generating feedback events (like "disable strategy" or parameter update suggestions) that update the Strategy and Signal modules. All significant events – signals, decisions, orders, risk triggers, and actions – are logged by the Compliance Logging module. This event-driven design ensures each component operates independently yet in sync with the others through well-defined messages. [1]

Message Bus and Event-Driven Workflow

At the core of the architecture is the **asynchronous message bus** that glues all modules together [2]. This could be a simple in-memory pub/sub mechanism or a message queue (like RabbitMQ or even Python asyncio events). The event-driven workflow is as follows:

1. **Market Data Event:** When a new market tick or bar is received by Data Ingestion, it creates an event (e.g., `MarketEvent(symbol, price, time)`). This event is published on the bus. Multiple subscribers pick it up: Signal Intelligence (to update indicators), Risk Engine (to update P/L calculations), possibly Macro Sync (to monitor volatility), and GUI (for price display).

2. **Signal Event:** Signal Intelligence processes the new data and if conditions are met, emits a `SignalEvent(strategy, symbol, action, confidence, context)`. This event goes to Strategy Engine (and could also be logged or monitored).

3. **Order Event:** Strategy Engine receives the signal, checks everything, and decides to trade. It then emits an `OrderEvent(strategy, symbol, volume, order_type, price, sl, tp)` representing the intention to execute a trade. This goes first to the Risk Engine.

4. **Risk Check:** Risk Engine intercepts the `OrderEvent`. It evaluates it – if it violates rules, it might modify or cancel the order. For instance, if volume too high, it might replace it with a smaller volume order event, or emit a `RiskAlertEvent(reason="OrderBlocked", details...)` and drop the order. If approved, it tags it as approved and passes it along (or simply does nothing, allowing it forward). The Execution Engine then receives the (possibly modified) OrderEvent.

5. **Execution & Trade Event:** Execution Engine takes the order, sends it to MT5. On success, it emits a `TradeEvent(strategy, order_id, fill_price, volume, status)` (and if position opens, maybe a `PositionEvent`). If an order was partially filled or pending, it might emit updates as well.

If order failed, it emits an event for that (which Risk or Strategy could use to react). Execution also emits events when positions are closed (by stop or manual) and when take-profits hit.

6. **Position/Risk Update:** Risk Engine listens to Trade/Position events to update exposure and P/L. It might emit a `DrawdownEvent(percent=X)` periodically or a `KillSwitchEvent` if a limit hit (which Strategy and Execution will act on immediately – e.g., Execution closes trades, Strategy stops trading).

7. **Macro Events:** Macro Sync emits events like `TradingHaltEvent(start, end, reason)` before a news, and `TradingResumeEvent` after. Strategy Engine and Execution subscribe to these – on halt, Strategy pauses new signals and perhaps Execution holds fire on new orders. Also, if Macro detects a flash crash, it could emit `VolatilityAlertEvent(level=high)` which Risk Engine might treat as a signal to tighten stops or freeze trading.

8. **Pattern/ML Feedback:** Periodically or after a set of trades, Pattern Detection might emit `StrategyUpdateEvent(strategy, new_param=X)` or `StrategyStatusEvent(strategy, disable=True)` if it finds a strategy underperforming. Strategy Engine would implement that (e.g., stop that strategy, or adjust its internal parameter). Or it could emit `MetaSignalFilterEvent(model=XYZ)` to tell Signal module to start using a new ML model to filter signals.

9. **GUI Interaction:** If a user presses "Stop All", GUI sends a `ManualControlEvent(action="pause")` which could be picked up by Strategy Engine (to pause trading) and Risk Engine (to perhaps also disallow until resumed) – effectively same as Macro's halt event. If "Close All" pressed, GUI sends an event to Execution to close positions and to Strategy to not open more. Strategy might also broadcast something like `StrategyToggleEvent(name, on/off)` when user toggles a strategy via GUI, which Strategy Engine handles.

10. **Logging:** The Compliance Logger subscribes to *all events.* It writes them to persistent storage in chronological order. Especially, it listens for OrderEvents, TradeEvents, RiskAlertEvents, etc., to log the critical info each contains [15] [16]. It may ignore less critical ones like every MarketEvent to avoid verbosity, focusing on decisions and outcomes.

This event-driven approach yields a resilient, extensible system: new modules can be added by simply subscribing/publishing relevant events (for example, one could add a Machine Learning predictive module that listens to MarketEvents and publishes its own SignalEvents, and the rest of system could take them if configured to do so). The loose coupling means each module can be developed and tested independently by feeding it the events it expects. It also makes the **backtester** easier – the backtester can drive the event flow by replaying recorded MarketEvents in sequence and observing the resulting cascade of events through the virtual system, exactly as in live trading.

Smart Risk/Compliance Overlays (FTMO Rules Implementation)

Adhering to FTMO's rules is non-negotiable; the system's risk overlay is built to strictly enforce these constraints in real-time, with multiple layers of protection. The **Max Daily Loss (5%)** and **Max Total**

Drawdown (10%) are implemented in the Risk Engine as described, but here's how the overlay functions holistically:

- **Pre-Trade Checks:** Before any trade is executed, the potential worst-case loss of that trade (assuming it hits its stop-loss) is added to the current day's realized loss + current open unrealized loss. If that sum would exceed 5% of starting balance for the day, the Risk Engine will not allow the trade. Similarly, it tracks the highest equity reached since account start to enforce 10% from peak – if new trade could push beyond, block it. This guarantees we don't *initiate* a trade that immediately violates rules.

- **Ongoing Monitoring:** At any moment, the Risk module computes **current drawdown** = (peak equity - current equity) / peak equity. If this crosses, say, 9% (close to 10%), it issues warnings (maybe GUI turns red). If it hits 10%, the system **immediately closes all positions** to cap the loss. For daily, it does similarly relative to day start equity. These triggers are automated and do not rely on any manual action.

- **Kill-Switch Coordination:** When a kill-switch event triggers (due to these limits or e.g. user panic), it's handled atomically: Risk Engine flags trading halted, Execution engine flushes all orders and closes positions at market, Strategy Engine stops generating signals. The GUI is updated to show "Trading Halted - Limit Reached". The system will not resume until the next day for daily loss (or manual reset). For the total drawdown, that might mean the account is essentially done (prop firms would typically stop the account if 10% breached).

- **Audit Trail for Compliance:** Every time these risk rules are engaged or tested, it's logged. E.g., *"Pre-trade check: trade risk $500, daily loss so far $1000, max allowed $1250 – OK"*, or *"Daily loss limit hit: equity fell from $100k to $94,950 (5.05%), closing positions"*. This way, if there's any dispute (say a trade closure slipped loss slightly over 5%), we have evidence of intent and execution times.

- **Secondary Limits:** The risk overlay also includes other prop firm style rules like not exceeding a certain lot size or number of positions. FTMO doesn't impose instrument-specific limits beyond drawdown, but we self-impose no more than, say, 5 concurrent trades and maybe no trade with >2% account risk. These are encoded similarly and provide extra safety margin.

- **No Cheating the Rules:** The system is programmed to treat these limits as absolute. For instance, it won't try to game the daily loss by hoping floating losses won't count – it counts open losses immediately as if closed (FTMO does consider open losses towards the loss limit). It also won't attempt risky maneuvers like martingale doubling down after losses because the risk engine would flag that as too much exposure increase after a loss.

In summary, the risk/compliance overlay operates like a strict overseer integrated in the code: it actively prevents rule breaches and would rather miss trades than come close to violating a limit. This is a disciplined, safety-first approach, aligned with the idea that *preservation of trading capital and adherence to prop firm rules is paramount over squeezing extra profit*.

Through the combination of architecture, event-driven coordination, and these comprehensive module designs, the GENESIS algorithmic trading system is engineered to be **modular, transparent, and resilient**. Each module's responsibilities and interactions are clearly defined, allowing institutional-grade reliability

and easier collaboration between human developers and AI assistants. By following the structured prompts and best practices outlined in this handbook, a team of engineers – human or AI – can effectively develop, test, and enhance GENESIS's capabilities while maintaining strict compliance and robust performance.

---

1 2 3 4 5 6 7 8 9 10 11 12 13 14 Comprehensive GENESIS Roadmap Research.pdf
file://file-CJpbvC6x8ro7tCrfwSiKNC

15 16 17 18 20 21 22 23 24 Building an Institutional-Grade MT5 Algorithmic Trading System.pdf
file://file-AW43SsUZ4PneFaNKZg3r3t

19 Upgrading __GENESIS__ to an Institutional-Grade Algorithmic Trading System.pdf
file://file-Az61hiMCLiNU2J4TQBcRov